EXHIBIT B

Dispatcher High Level Design

Revision 1.30

April 30, 2002

Simon Knee

Revision History

Revision	Date	Author	Description of Changes
0.10	March 13, 2001	Simon Knee	Initial document.
0.11	April 3, 2001	Simon Knee	Removed Flow Director CAM section into its own HLD document. This is document is still in its initial phases.
0.12	April 9, 2001	Simon Knee	Snapshot after Pramod took a version for initial review.
0.13	April 16, 2001	Simon Knee	Snapshot after Pramod took another version for review.
0.14	April 19, 2001	Simon Knee	Snapshot after Brian took a preview version.
1.00	April 30, 2001	Simon Knee	Various updates after review including numerous typos and clarifications, added section on blocking events and Denial of Service, added known issue about requirement for path from TCP core to Dispatcher. Charles Kaseff: Control Register definition was missing. Mark Feinstein: Added event queue and message bus to "Dispatcher Overview" figure. Robert Johnson: Numerous typos, fixes in flow chart logic and tables. Perry Virjee: Updated "Dispatcher Overview" figure to show that the message bus is really two shared buses. Brian Petry: Numerous typos, spelling, grammar, logic and clarifications. Pramod Argade: Stateless events still require FDC interaction. This HLD is now released for the current feature set.
1.01	September 27, 2001	Simon Knee	Updates from Perry Virjee and Robert Johnson, plus other fixes as described below. All section, figure and table numbers are relative to v1.01. • The *Process Deleting Entry Flow Chart* incorrectly stated that the Processor Core event has a workspace ID. • The Process Protocol Done Flow Chart had a branch for whether the event type required the LUC. However, this branch had already been made in the Process Event Flow Chart. This was incorrect: we need to check for done events in the Process Event Flow Chart. • The *Process Deleting Entry Flow Chart* incorrectly stated that the Processor Core event has an FDC index. • Modified the "Expected Performance" section to say that we do not expect all performance metrics to be met at the same time: they are disjoint. • Added a MASK register (section 3.14.9). • Updated the Dispatcher Overview of Figure 1. • Changed the Process Protocol Done Event (section 2.6.8) so that it correctly sends the LUC a Discard command. • Added a FDC In Timer State register (section 3.14.10), used in the flow chart of section 2.6.5. • Added the ECPARMS register (section 3.14.12). This has the Event Type to use for a SERVTIMER command (section 2.6.7), and the LUC command value to use for a Discard command (section 2.6.8). • Defined a region in the register map (section 3.14.1) that the FDC can use. Made it clear that the FDC and Dispatcher share a common MMC bus. • Updated section 1.5 on queue handling to make it explicit that the Dispatcher must be able to read a complete event without de-queuing it. • Made it clear that Done Events now require two 128-bit words. • Made it more explicit that the Command / Response values of Table 8 are only used for Dispatcher to Processor Core communication, and are not required for Done events.

Revision	Date	Author	Description of Changes
			• Renamed *Other Event Queue* to *ND Event Queue*. ND stands for Not Done. Made it clear that the Dispatcher must decode the address on a MBUS request and deposit the message in the correct queue. • Updated the bus descriptions of section 3 to allow for the SMC, two IPUs, and other more minor issues.
1.20	October 12, 2001	Simon Knee	• Made the CONTROL register read and write instead of write only (see section 3.14.1). • Modified section 1.4 to explain the *forwarding mode* of events. Also modified the Process Event Flow Chart to allow for the forwarding mode. Added flow charts for Stateless Unicast Event Processing and Drop Event Processing. Modified event parameter registers (section 3.14.16) so that a 2-bit FW_MODE field replaces the REQ_LUC and ~~DROP bits. Also modified the bit positions in that register.~~
			• Added section 2.4 on FDC responses. Explained why we continue to retry FDC command even if we got an error response. Modified flow charts so that if the FDC response with an error we do not drop the event – instead we go back to the Dispatcher start. • Modified the IPU Control Event (Figure 23) to allow for the Interface Core ID, Interface Workspace ID, Event Index and Event Mode. • Removed the DPARMS register and replaced with a DONE_TD bit in the EPARMS register (section 3.14.16). • Changed the USID and ULID LUC command formats to Flow Key Format. See section 3.1.1, Table 4. • Filled out the external signal description of the SMC Dispatcher Bus (see section 3.8.3). • Explained how Message Bus addresses are formed for Type A transactions (see section 3.6.2), and explained that Type B transactions do not have addresses created (see section 3.7.2). • Modified section 3.5.1 on the Control Event format to say that the Dispatcher only modifies the *Scratch Data Length* and *Running Checksum* fields for events received from IPU0 or IPU1. • For repeated stateless event processing the Dispatcher must issue an increment of the reference count on the Scratchpad. Added section 2.5.2 to explain this, and modified the stateless processing section (2.6.11) to use it. • The Dispatcher no longer forwards events when the FDC entry is found in the *DELETE* state. Instead it waits until the FDC entry is no longer in the *DELETE* state. This simplifies the design of the Dispatcher, and does not add any significant issues. Section 2.7 on Blocking Events was updated. • Changed address values in register map to be in hexadecimal. • Added the FDC_ERR register (section 3.14.5) and modified flow charts such that FDC error responses (5'b11111) increment this counter. • Added section 2.5 on the Dispatcher and Scratchpad interaction – for checksum information etc. Added Query Scratchpad subroutine (section 2.6.2). Modified flow charts so that the Scratchpad is queried after a good FDC response is received, i.e. just before we send the event to the processor core. • Made it clear in section 3.5.1 that the DERR bit should be copied from the input event, since it may already be set to zero. • Added section 1.4.1.1.1 that describes allocation rules for Event Masks. • Added section 3.14.13 on the MB_BPRESSURE register (Message

Revision	Date	Author	Description of Changes
			Bus Backpressure). • Added section 1.4.3 on split Protocol Core / Interface Core processing. • Modifications in a number of areas to allow for split Protocol Core / Interface Core processing. • Added section 1.4.4 to explain event type based forwarding, and to explain why if the most significant bit of the Event Type is set then we must be performing unicast stateless forwarding. Because of this fact, the number of Event Parameter registers (section 3.14.16) was reduced from 64 to 32.
1.21	October 22, 2001	Simon Knee	• The LFLK command was missing from Table 4: LUC Command Summary. • Tidied up the use of "assume" to make this HLD more concrete. • Attempted to make the usage of the EPARMS register block a bit clearer (see section 3.14.16). • Bandwidth requirements for IPU to Dispatcher buses were not quite correct in section 3.5.2. • Added section 2.3.2 on Core Indexes. • Figure 14: Process Timer Event Flow Chart incorrectly said that the FDC Index is taken from the FDC response when a Processor Core event is created. It comes from the Timer Event (LUC Command). • Figure 23: IPU Control Event Format had incorrect bit numbering for the Flow Key. Now fixed. • Added section 2.2 on example event processing. This illustrates a simple single core case, as well as a case where the Protocol Core issues a stateless done in order to quickly release and event index.
1.22	November 4, 2001	Simon Knee	Bug fixes, clarifications and minor changes: • Made it clear that the Event Mode is not examined for stateless done events. See section 3.5.1 on the IPU Control Event format. • Added interface disable bits to the CONTROL register (section 3.14.4). • Modified Figure 1: Dispatcher Overview to include the Message Bus and Global Timestamp for the LUC. • The *Issue LUC Command* boxes of Figure 13: Issue LUC Command Flow Chart and Figure 14: Process Timer Event Flow Chart did not include the FDC Index as something that is set. • The *Issue LUC Command* box of Figure 14: Process Timer Event Flow Chart should use LUCCM2 instead of LUCCM1. • Figure 15: Process Protocol Done Flow Chart, the case when the FDC entry is in the *UPDATE* state: The *Issue Command to LUC* box uses a Flow Key format, not Socket ID. • Added the FDC_DELST register. See Figure 12: Process Stateful Event and section 3.14.14. • Modified the ETSERV field of the ECPARAMS register (section 3.14.12) so that it is only 5-bits. This also required some modifications to section 2.6.7 and Figure 14: Process Timer Event Flow Chart. This was done since the timer event must be stateful, so the MSB must be zero. • Figure 11: Process Event Flow Chart was modified to clarify that *Unicast Stateless Processing* is used if the MSB of the Event Type is set. • Problem Report (PR) 57 fix: Figure 16: Process Drop Event Flow Chart was modified so that a LUCDISC command is not issue. Removed the LUCDISC field from the ECPARMS register (section 3.14.12). • PR 59 Fix: Figure 22: LUC Timer Event Format was modified so that it

Revision	Date	Author	Description of Changes
			includes the 22-bit Socket ID. Modified Figure 14: Process Timer Event Flow Chart so that a Socket ID LUC command format is used rather than Flow Key. Modified Table 2: Example TCP Dispatcher Configuration to show that the LUCCM2 command for a timer event should be LSID. This allowed the open issue on LFK vs. LSID for timer events to be closed. • Clarified an issue for repeated unicast where an increment by zero will be sent to the Scratchpad. The Scratchpad will treat this as a NOP. See section 2.6.11. • Modified the MB_BPRESSURE register (section 3.14.13) so that it contains two values: one for the Done Queue, and one for the ND Queue. • Added the DONE_ALMOST_FULL and ND_ALMOST_FULL signals to Figure 1: Dispatcher Overview. • Made it clear that the Scratchpad reference count increment is used for events from the SMC and the ND queue, as well as events from IPU0 or IPU1. • Modified Figure 10: Dispatcher Start Flow Chart and Figure 11: Process Event Flow Chart to show that the DERR bit is only examined for events received from IPU0/IPU1. Also re-arranged these flow charts so that Timer Events are immediately dispatched to the *Process Timer Event* flow chart, and Done events are immediately dispatched to the *Process Protocol Done Event* flow chart. • Expanded section 2.1 and renamed it to *Minimal TCP Configuration*. The various forms of Done Events were described, as well as how to use the new RELS command (release) of the LUC. • Added section 3.15 on how to initialise the Dispatcher. • Added section 2.2.4, Figure 5: Example: Dual Mode Stateful Event, Flow Not Found, to illustrate the use of the RELS LUC command.
1.23	November 11, 2001	Simon Knee	Bug fixes, clarifications and minor changes: • PR 68 Fix: Renamed the MB_BPRESSURE register (section 3.14.13) to BPRESSURE. Added a LUC_ELMNT field to that register – this defines the number of elements in the *LUC Request Queue* before the Dispatcher disables all input interfaces. • Section 3.7.3, Done Event Queue size. Increased from 15 to 20 to match the RTL. • Section 2.6.7, *Process Timer Event*. Made it clear that the Socket ID should be set to zero in the Processor Core event. • Figure 12: Process Stateful Event said "Increment FDC In Timer State reg[FDC_DELST]. It should have said "FDC in Delete State". • Clarified the usage of the FDC_ERR bit in the STATUS register. Modified Figure 12: Process Stateful Event, Figure 14: Process Timer Event Flow Chart and Figure 15: Process Protocol Done Flow Chart to clarify that the FDC_ERR bit is used. • PR 76 Fix: Added bits to the status register that allow queue backpressure and overflow to be detected. • Added the SP_CNT_BUS to Figure 1: Dispatcher Overview. This is used solely for reference count increments. Section 2.5 on *Dispatcher and Scratchpad Interaction* was modified to use this bus. This was an omission in the HLD: the RTL already works this way. • Removed the *FDC Request Queue* and *FDC Resp. Queue* from Figure 1: Dispatcher Overview as they do not exist and are not required.

Revision	Date	Author	Description of Changes
1.24	December 16, 2001	Simon Knee	Bug fixes: • PR 87 Fix: FDC_DELST register should be read-only in register map (section 3.14.1). • PR 124 Fix: STATUS register was missing an INV_MMC_ADDR bit. See section 3.14.3. • PR 116 Fix: Made it clear that a CREPEAT value of zero is an invalid configuration. See sections 2.6.11 and 3.14.16. • Added default values to all registers of section 3.14. • PR 154 Fix: Description of LUC to FDC and FDC to LUC busses was missing. Added sections 3.12 and 3.13. • PR 196 Fix: Added the DEBUG register (see section 3.14.15). • Added section 3.14.2 that explains that the Dispatcher must process a register write in seven clock cycles or less. • Section 3.14.4 (CONTROL register). Made it clear which interfaces should / should not be disabled while the Dispatcher is running. • PR 197 Fix: ND queue removed. Any associated bits in registers changed to N/A. • PR 198 Fix: SMC_DISP_Almost_Full signal added, servicing of IPU0/1 disabled if this signal is set. ND queue removed as in PR 197 above. • PR 244 Fix: Figure 4: Example: Dual Mode Stateful Event, Early PC Event Release was corrected to show that it uses a *Done Event, Stateless*.
1.25	December 21, 2001	Simon Knee	Bug fixes: • Numerous typos found by Nassir were fixed. • PR 256 Fix: Modified Figure 4: Example: Dual Mode Stateful Event, Early PC Event Release to show that the *Event Mode* is not used for a stateless done. Modified the text of that section to explain how the Protocol Core ID field is always used for stateless done events, i.e. an Interface Core must overwrite this field before issuing a stateless done.
1.26	January 22, 2002	Simon Knee	Bug fixes: • Added a note to *Table 7: IPU Control Event Field* Descriptions about who sets the *Event Size* field. • PR 347 Fix: DEBUG register description (section 3.14.15) modified. BYPASS bit is no longer used. • PR 377 Fix: Exactly when to set the watermarks in *Table 23: Message Bus Backpressure Register Bit Definitions* was clarified. • Closed the last remaining issue on timer processing being blocked out by back-to-back packets. See section 2.7.1, step 4.
1.27	February 13, 2002	Simon Knee	Bug fixes: • PR 124 Fix: INV_MMC_ADDR bit of STATUS register (section 3.14.3) is only set for invalid reads, not invalid writes. • PR 452 Fix: Modified section 2.6.2 to make it clear that the *Running Checksum* and *Scratch Data Length* fields of an input event are not modified unless the event comes from IPU0/1. • PR 457 Fix: DEBUG register (section 3.14.15) adjusted to 6-bits. • PR 579 Fix: Adjusted the name of some registers so they are consistent with the rest of the chip.
1.28	February 28, 2002	Simon Knee	Bug fixes: • PR 616 Fix: Modified sections 3.5.1 and 3.6.1 to say that Done Events must have a length of exactly two 128-bit words.
1.29	March 18, 2002	Simon Knee	Bug fixes: • PR 942 Fix: Added a 12-bit sequence number to *Figure 23: IPU Control Event Format*, and added section 3.6.4.

Revision	Date	Author	Description of Changes
1.30	April 30, 2002	Simon Knee	Bug fixes: • PR 1220 Fix: Modified *Table 21: Mask Register Bit Definitions* to show which bits were spare. • PR 1225 Fix: Section 2.5.1.2 added on *Checksum Length Issues*.

Table of Contents

List of Figures

List of Tables

1 Introduction

In this document we describe the high level operations of the Dispatcher. This document should describe all the necessary behaviour of the Dispatcher, without enforcing any particular implementation method.

1.1 Related Documents

Document	Revision	Author
Astute Content Processor Architecture Presentation	N/A	Fazil Osman
Deadlock Analysis and Avoidance	1.20	Brian Petry
Event Specification	1.22	Simon Knee
Flow Director CAM (FDC) High Level Design	1.32	Simon Knee
Host Message API (Level 3) High Level Design	1.23	Billy Oostra
Input Processing Unit (IPU) High Level Design	1.37	Bob Sefton
LookUp Controller (LUC) High Level Design	1.29	Simon Knee
ManageMent Controller High Level Design	1.11	Nitesh Mehta
Message Bus High Level Design	1.30	Mark Feinstein
Output Processing Unit (OPU) High Level Design	1.24	Bob Sefton
Packet Processor High Level Design	1.20	Octera
Protocol Cluster High Level Design	1.16	Kirk Larson
Queuing Model Trace of a Simple HTTP Request	1.00	Simon Knee
Socket Memory Controller High Level Design	1.12	Simon Knee
Some Statistical Plots of Web Traffic	1.00	Brian Petry
Scratchpad High Level Design	1.50	Charles Kaseff
SPI-4 Module High Level Design	1.02	Bob Sefton
TCP Processing Paths for the Content Processor	1.00	Simon Knee, Brian Petry
TCP/IP Algorithm Review	0.91	Brian Petry
TCP/IP Core Software: Functional Specification and Conformance Statement	1.10	Brian Petry
TCP/IP Core Software: High Level Design	WIP	Brian Petry

1.2 Overview

As illustrated by Figure 1, the Dispatcher processes events from one of two IPUs, the SMC, the Processor Cores and the LUC. All packet events, host events and timer events go through the Dispatcher. After Dispatcher processing, events are then moved to a Processor Core using the Message Bus. When the Processor Core has finished processing the event, it issues a done event to the Dispatcher using the Message Bus[1].

Events that are received from an IPU interface have been formatted by a Packet Processor. Multiple Packet Processor parse the same IPU stream, so the IPU employs special techniques to ensure that order is preserved. Note that it is not the responsibility of the Dispatcher to perform this re-ordering.

Note how in Figure 1 both the Dispatcher and the FDC share the same ManageMent and Control (MMC) interface. The Dispatcher includes a small block, the *Slave MMC Mux*, for distributing this bus to both the main Dispatcher block and the FDC. For this reason, all of the FDC registers are actually mapped into the Dispatcher. See section 3.14.17 for further details.

[1] The Message Bus will actually consist of two buses: one for each direction into and out of the Dispatcher. This bus is shared by a number of devices: each device, or possibly each queue is assigned an address. See the Message Bus HLD for further details.

The Dispatcher has one queue from the Message Bus that is used for *Done Events*. This queue has the capability to assert backpressure to the Message Bus, which indicates that a new event should not be sent (the existing event is allowed to complete).

Note that it is not possible for the Dispatcher to drop an event. Any event that is to be dropped is marked as "please drop" and then forwarded to a processor core. The reason for this is that dropping an event complicates the design of the Dispatcher.

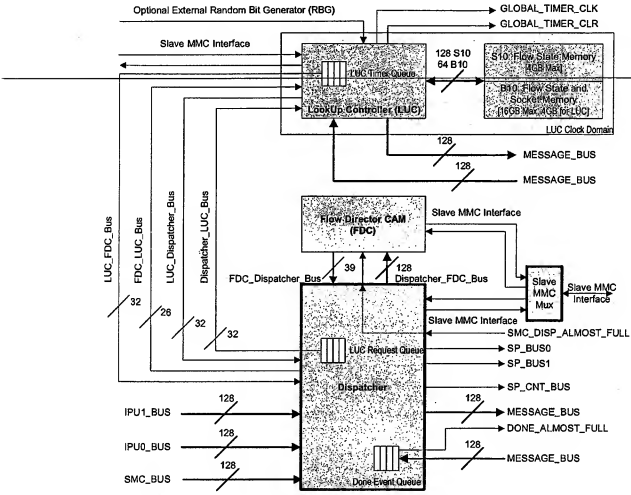

Figure 1: Dispatcher Overview

This document relies heavily on the features described in *Flow Director CAM (FDC) High Level Design*. If the reader is not familiar with this document then it is highly recommended that they read it. While reading the Dispatcher HLD you will often need to refer to the FDC HLD.

1.3 Dispatcher Input Events

There are four major event types that the Dispatcher will process: packet events, timer events, host events and done events. Figure 1 indicates where such events may come from.

Each of these major event types can also be split down into sub or minor events. For example, the major event may be *stateful packet event*, but the minor event type is *TCP*. Other examples of a packet event include UDP, ICMP, ARP, TCP-WWW, TCP-FTP etc. In fact, as the Astute Content Processor is applied to new vendor applications it is likely that the number of event types will increase[2].

1.4 Processing Events

1.4.1 Forwarding Modes

The events that the Dispatcher processes can be split into two groups: those that are stateful (require the LUC), and those that are stateless (do not require the LUC). Stateless events can be subdivided again, into unicast, repeated unicast and drop. In fact, events that need to be dropped are simply unicast stateless events with a special flag set. Based on the Event Type, the Dispatcher determines the forwarding mode of an event. The forwarding mode is one of four values as listed below and described in more detail in the following sections:

1. Stateful Event Processing
2. Unicast Stateless Event Processing
3. Unicast Stateless Event Processing with Drop
4. Repeated Unicast Stateless Event Processing

1.4.1.1 Stateful Event Processing

Some event types, but not all, will require interaction with the LUC so that the state of a flow can be retrieved. An example of such an event type would be a *TCP Packet Event*. This will require the LUC to find the TCP flows state and to pass this onto the assigned Processor Core. The FDC is used to ensure that the same Processor Core is used for other events in that flow.

To allow some flexibility the Dispatcher can be programmed, via registers, such that each event type has its own set of LUC commands. For each event type we can specify two LUC commands. Which LUC command to use depends upon the value of a special *create* flag in the event header. A typical TCP event configuration would be to use the *lookup with flow key and create* and *lookup with flow key and no create* LUC commands. However, in some scenarios we may never want the LUC to create flows automatically, so both commands would be *lookup with flow key and no create*.

1.4.1.1.1 Stateful Event Type Based Forwarding

As described in the *Flow Director CAM HLD*, the FDC selects which Processor Cores can process an event based upon the Event Type. This allows different cores to be nominated for processing different types of events – perhaps that is to help distribute the load, or perhaps it enables different software images to be loaded onto different processor cores. For further information on Event Types and Event Masks, including the event type sub-structure, the reader is directed to the *Flow Director CAM HLD*.

Note that once an entry has been allocated in the FDC for a stateful event, any events that are received with that flow key will be forwarded to the specified processor cores, regardless of the event type. That means that once an FDC entry has been allocated, the Event Mask is ignored. For this to make sense the software should follow this rule:

1. Let F be a flow key.
2. Let {T} be the set of Event Types that *can* be assigned to that flow key.
3. Let {M} be the set of Event Masks that are programmed into the FDC for each member of {T}.
4. It must be the case that all members of {M} are identical.

[2] Although the number of event types may increase over time, not all applications will be required to process all events. There is also the concept of a "catch all" event that reduces the total requirement for the number of unique event types.

If the above *were not* true then the following could occur:

1. Dispatcher receives an input event for flow key F, Event Type T_1, and the FDC allocates it to a Processor Core P_1.
2. Dispatcher receives an input event for flow key F, Event Type T_2. It just so happens that P_1 is not included in the Event Mask of T_2. However, since the entry was found in the FDC the Dispatcher will still forward it to P_1.

1.4.1.2 Stateless Event Processing

Stateless event processing occurs when an event type does not require interaction with the LUC. An example of such an event type would be an *ARP Packet Event*. In this case there is no state associated with each ARP frame, so we simply pass the event on directly to the Processor Core. This stateless event can either be unicast to a single core, or sent to multiple cores, as described below.

1.4.1.2.1 Unicast Stateless Event Processing

For this type of stateless event processing we wish to allocate a processor core from a set of capable processor cores. This set of capable cores is based upon the value of the Event Type. One of those cores is selected, and the stateless event is forwarded to it. This is useful for sending events to a single core, e.g. perhaps ICMP events.

1.4.1.2.2 Unicast Stateless Event Processing with Drop

As described in section 1.2, the Dispatcher is not capable of dropping events. To overcome this the Dispatcher has the concept of forwarding an event to a processor core just for the purpose of dropping it. In fact, when it does this it marks the event in such a way that the processor core knows it must be dropped. The mechanism used to forward the event is exactly as described in section 1.4.1.2.1 above.

1.4.1.2.3 Repeated Unicast Stateless Event Processing

Note that for stateless event processing we may want to pass the same event onto multiple Processor Cores. An example of this is if the software architecture was such that each Processor Core had its own route / ARP table. In this case we would need to multicast *ARP Packet Events*: such events may be an ARP reply, in which case each Processor Core may want to see the response[3]. Other types of packet events may also require multicasting to a subset of the Processor Cores.

To allow maximum flexibility, if an event type is marked as not requiring a LUC interaction, then the Dispatcher contains a bitmap of Processor Cores that should receive this event. This bitmap allows us to send the event to programmable groups of Processor Cores.

The method that is used for multicasting events is repeated unicast, i.e. if the Dispatcher determines that the event must be sent to N Processor Cores then the Dispatcher will perform N unicast operations, sending the event to each Processor Core in turn. A faster method would be to multicast the event in hardware, but this requires extra support from the Dispatcher and Processor Cores. Since multicast frames are not expected to be in the fast path, we can use the repeated unicast method.

An issue with repeated unicast is what we should do if we need to send an event to one or more Processor Cores, but none of those Processor Cores have space in their event queues. We cannot simply discard the input event just because there is no space. Instead we must leave the input event at the head of its queue, and try to finish the repeated unicast at a later date. This has the consequence that such an event can block the head of a queue, even though there may be (unicast) events behind the head that could be serviced. We must also record which Processor Cores have been sent the repeated unicast, and which ones are left. We must record this on a per input queue basis, since it is possible that the event at the head of each input queue can be performing a repeated unicast.

[3] If any Processor Core can issue ARP request then there is no way for the Dispatcher to know which core to send the ARP response to.

1.4.2 Timer Event Processing

In the Astute Content Processor the Timer Control Unit, TCU[4], is responsible for keeping per flow timers. If such a timer expires then a timer event is issued to the Dispatcher. These events are complicated by the fact that a packet or host event **must** never be serviced before a timer event. The reason is that the packet or host event could cause the timer event to be cancelled, but once the LUC issues the timer event there is no way for the Processor Core to invalidate it.

Due to this requirement the Dispatcher may process a non-timer event only to find that the FDC says it is in the *TIMER* state. This is indicating that the LUC has expired the timer, and that the event is in transit from the LUC to the timer queue on the Dispatcher. In this case the Dispatcher must stop processing the current event, and change to processing the timer event in the timer queue.

For further information on how timer events are ordered the reader is directed to the FDC HLD.

1.4.3 Protocol Core and Interface Core Forwarding

The Dispatcher supports the concept of forwarding events to either the Protocol Core or the Interface Core. Most of the implementation of split Protocol Core / Interface Core handling is hidden by the Flow Director CAM. The Dispatcher simply forwards the event to the Processor Core that the FDC indicates. To do this the FDC uses the Event Mode field of the FDC response. Table 1 illustrates the values of the Event Mode in the FDC response. Note that in FDC response only values 2'b01 and 2'b10 are valid[5].

Event Mode Value	Description
00	Event Index does not describe a protocol core or interface core.
01	Event Index is for a protocol core.
10	Event Index is for an interface core.
11	Invalid.

Table 1: Event Mode Values

1.4.4 Event Type based Forwarding

Each Processor Core may be capable of processing a certain protocol and not others, e.g. TCP may be on cores 0 through 15, but ICMP is only on cores 14 and 15. To solve this problem the Flow Director CAM uses the *Event Type* as an index into an array of bitmaps where each bitmap represents which Processor Cores are capable of processing this event. Note that it is a Packet Processor in the IPU that assigns the 6-bit Event Type.

The Dispatcher also keeps information on a per Event Type basis. For example, the forwarding mode described in section 1.4.1 above is determined on a per Event Type basis. This requires the Event Parameter registers block, where each Event Parameter register contains information with a given Event Type.

1.4.4.1 Event Type Sub-Structure

The Event Type also has sub-structure that allows the Dispatcher to track 32 registers instead of 64. This sub-structure is such that if the most significant bit (bit 5) of the Event Type is set, then by definition a stateless unicast event is being processed[6]. If a stateless unicast event is being processed then no field in the Event Parameter register is required. If the most significant bit (bit 5) of the Event Type is not set, then

[4] The Timer Control Unit (TCU) is part of the LookUp Controller (LUC). In fact, the TCU and LUC are so closely integrated that they can be considered the same device.
[5] In fact, the Dispatcher only checks for 2'b01 and assumes it is 2'b10 otherwise.
[6] The Flow Director CAM also examines bit 5 of the Event Type. If bit 5 is set then the lower 5-bits of the Event Type contain a Core ID. The event is then forwarded to that Core ID. See the *Flow Director CAM HLD* for further details.

we use the lower 5-bits as an index into the Event Parameter registers block. Dependent upon the forwarding mode found in the Event Parameters block, we then process the event appropriately.

1.5 Queue Handling

In the above sections we have assumed that the Dispatcher can read a queue element without actually de-queuing the element. This is required for two reasons:

1. So that we can attempt to service an event, but then back out if we find we cannot continue.
2. It is required for repeated unicast, where the same event is sent to multiple destinations.

1.6 Processor Core and LUC Required Behaviour

In order for the Dispatcher to function correctly, the Processor Cores and LUC must behave as follows:

1. The LUC clears the valid bits of the workspaces before it issues a RMFIDX command to the FDC. If the LUC sees that the response to the RMFIDX command indicates that the FDC entry is in the *RECEIVED* state, then the LUC sets the valid bits again.
2. Upon completion of processing of an event, the Processor Core clears the event's valid bit. It does this before sending the done event to the Dispatcher.

2 Functional Operation

The following sections describe how the Dispatcher processes the various events that it receives, and how it uses the Flow Director CAM (FDC) to ensure coherency.

2.1 Minimal TCP Configuration

The Dispatcher uses registers so that the actual event type values of TCP packet events, timer events, etc., may be programmed at run time. While this programmability makes the Dispatcher relatively flexible, it makes the explanation of its operation more confusing. For this reason this section defines a minimal configuration that would be required to make TCP termination work.

For TCP termination we must at least configure event types for TCP packets, done events and host events. Table 2 illustrates the settings for some event types in the Event Parameter registers of the Dispatcher (see section 3.14.16). This is not an exhaustive list of event types required for TCP, and it is likely that more event types will be required. In this table an X represents don't care. It is not expected that the reader understand (or even read) this table on the first pass of this document. Instead it can be used as a reference when following the flow charts, and it should help show the expected path / commands for TCP termination.

Event Type	Forwarding Mode [FW_MODE]	Use Flow Key [LUSEFK]	Event is Teardown [DONE_TD]	LUC Command 1 [LUCCM1]	LUC Command 2 [LUCCM2]
TCP Packet	Stateful	Yes	X	LFLKC	LFK
Done Update Event [Flow]	Stateful	X	No	TDFK	USID
Done Teardown Event [Flow]	Stateful	X	Yes	TDFK	USID
Done Update Event [Listen]	Stateful	X	No	TDLK	ULID
Done Teardown Event [Listen]	Stateful	X	Yes	TDLK	ULID
Done Release Event [Flow or Listen]	Stateful	X	Yes	RELS	RELS
Done Event, Stateless	Unicast Stateless	X	X	X	X
Timer Event, Host Event: read, write, close	Stateful	No	X	X	LSID
Host Event: create socket	Stateful	Yes	X	LFKC	X
Host Event: listen registration	Stateful	Yes	X	LLKC	X

Table 2: Example TCP Dispatcher Configuration

Some interesting points to note from Table 2 are:

1. Host events should always use the Socket ID for LUC commands (LSID) where possible. The exception is the host event that asks for a socket to be created: in this case we use a lookup with create command (LFKC) using the Flow Key, since the Socket ID does not yet exist[7].

2. For Done Update Events the LUC also uses the Socket ID to get direct access to the flow state. However, the Dispatcher does not use the Socket ID variant of the LUC command: instead the LUC gets the Socket ID directly from the workspace of the Processor Core.

3. Done events require multiple event types. The different event types indicate if this is an update, teardown, listen / flow, release etc. It is important to note that the Processor Core must re-write the Event Type of an input event before it sends it back to the Dispatcher as a Done Event.

4. The Done Release Event is used when the Dispatcher requested a lookup, but no flow state was found or created by the LUC. In this case the LUC must be told to release any resources it has. The Done Release Event must set the *Event is Teardown* flag (DONE_TD) to 1. This causes the FDC entry to go into the *DELETE* state, which prevents the forwarding of events while the LUC is executing the RELS command. This makes the logic for the RELS command simpler for the LUC.

2.2 Example Event Processing

In this section we illustrate the processing of a TCP event from the viewpoint of the Dispatcher. This section requires some understanding of how the Dispatcher works. For the first read of this document you need not fully absorb this section. However, it is worth reading on the first pass since it provides insight into the operation of the Dispatcher. It is recommended that you re-read this section when you are more familiar with the operation of the Dispatcher.

2.2.1 Single Core Mode, Stateful Event

In the first example, Figure 2, the FDC is in the Single Core mode. The Dispatcher receives an event from the IPU, issues an LFKCREATE to the FDC[8] and then forwards an LFK (Lookup Flow Key) command to the LUC. The Dispatcher then forwards the input event to a Protocol Core. When the Protocol Core receives the workspace and the event, it will perform TCP processing. It then writes back the workspace and issues a stateful done event to the Dispatcher. In response to this done event, the Dispatcher issues an UPFIDX command to the FDC, and assuming this is the last outstanding event for that flow, issues a USID (Update with Socket ID) to the LUC. After the LUC has processed the USID command, it issues an RMFIDX to the FDC. This removes the FDC entry and de-allocates the Workspace ID.

[7] This is an active open, as opposed to a passive (via listen socket) open.
[8] We do not show responses from the FDC since in these examples the response is always valid.

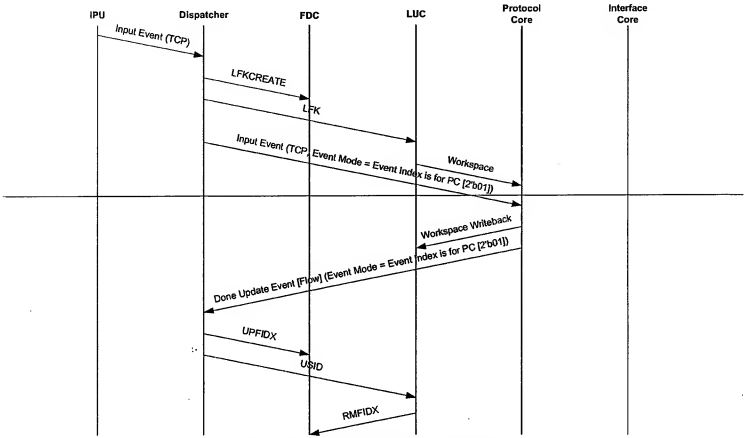

Figure 2: Example: Single Core Mode, Stateful Event

2.2.2 Dual Core Mode, Stateful Event

In Figure 3 we illustrate the same scenario as above, but this time the FDC is in the Dual Core Mode. We also assume that in this particular example, that the Protocol Core will receive the event from the Dispatcher, and that it will forward an event to the Interface core via an inter-core message.

The processing of Figure 3 below is exactly the same as the single core example up to the point at which the Protocol Core receives the input event. In the dual core example the Protocol Core then performs TCP processing, updates and writes back its workspace, and then forwards an event to the Interface Core. The Interface Core then performs some processing, updates and writes back its workspace, and then sends a stateful done event to the Dispatcher. The Dispatcher then follows the same steps it did in the single core example.

Note that in the time between when the Dispatcher sent the Input Event and the time it processed the stateful done event, an event queue element in the Protocol Core was marked as busy. However, the Protocol Core had finished processing that event queue element as soon as it sent the inter-core message, i.e. a resource was allocated for longer than it really needed to be. This problem is amplified if the Interface Core takes an excessive amount of time to send the stateful done event. This problem is addressed in the next example.

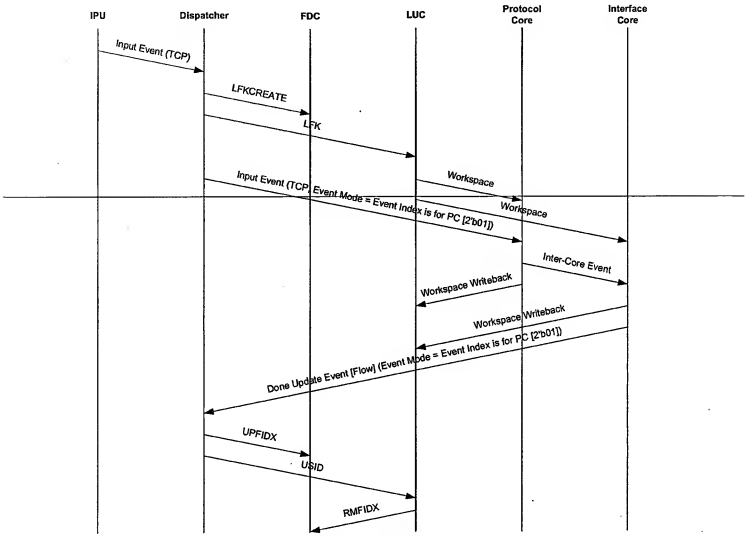

Figure 3: Example: Dual Core Mode, Stateful Event

2.2.3 Dual Core Mode, Stateful Event, Early PC Event Release

In the example illustrated in Figure 4, we perform the exact same processing as the dual core example above. However, in this example we want the Protocol Core to release its event queue element as soon as possible. The Protocol Core does this by issuing a stateless done event after it finished processing it's input event. The only purpose of this stateless done event is to release an event queue element. Since the event type is stateless the Dispatcher processes this done event by sending a RELEVENT to the FDC, effectively marking the event queue element as available.

Note that a stateless done event **always** uses the Protocol Core ID to determine which Processor Core to release the event from. In this example it is the Protocol Core that is doing the stateless done, so Protocol Core ID field need not be changed from the input event. However, if the Interface Core were performing the stateless done then it **must** overwrite the Protocol Core ID field with the Interface Core ID.

Note that when the Interface Core sends the stateful done event we need to make sure that it does not attempt to de-allocate the Protocol Core's event queue element[9]. The Interface Core does this by changing the Event Mode of the done event so that it has value 2'b00, i.e. it does not release an event queue element.

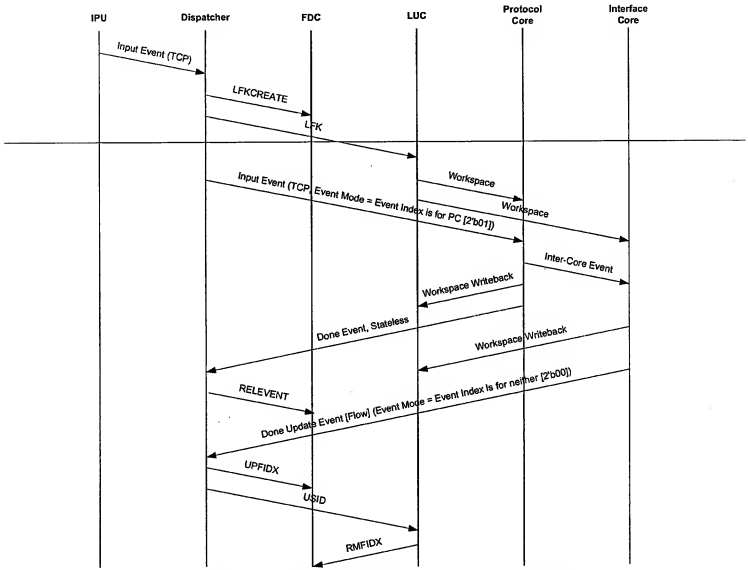

Figure 4: Example: Dual Mode Stateful Event, Early PC Event Release

2.2.4 Dual Core Mode, Stateful Event, Flow Not Found

Figure 5 illustrates the case where the LUC does not find a flow state, and the LUC command is such that it is not allowed to create one. This same scenario also occurs when the LUC attempts to create a flow but cannot due to a lack of resources, Termination Access Control List denial, or because the hash list would grow too long.

[9] This event queue element may have been re-allocated.

The first few steps of Figure 5 are the same as previous examples. The difference starts when the LUC sends the workspaces to the Protocol and Interface core with a response code that says *Not Found*. When the Protocol Core receives the input event from the Dispatcher, it performs the necessary processing, e.g. it sends a reset to the TCP peer. However, it now has a Workspace ID and an Event Index that it must de-allocate, but it cannot cause an update (USID) or teardown (TDFIK) command to be sent to the LUC, since the flow does not exist. This is where the *Done Release Event* is used. This signals to the Dispatcher, via the Event Type, that a TDFIDX should be issued to the FDC and that the Event Index can be released. When appropriate, the Dispatcher then sends a RELS command to the LUC. The LUC will de-allocate any internal resources, and then send an RMFIDX command to the FDC, effectively releasing the Workspace ID.

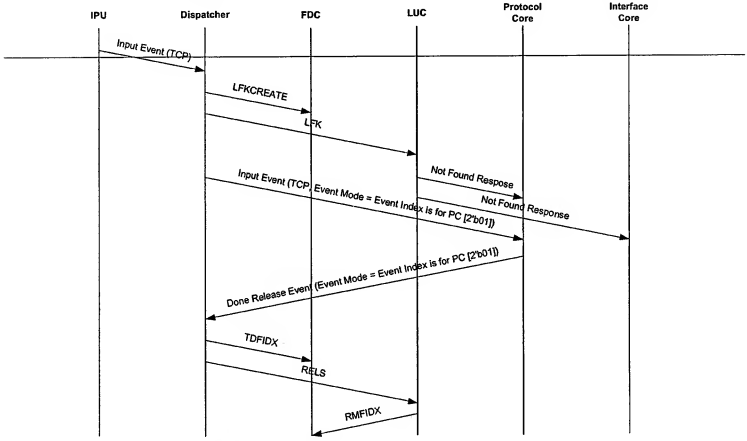

Figure 5: Example: Dual Mode Stateful Event, Flow Not Found

2.3 *Number Schemes*

In the following sections we describe the numbering schemes used for the Core ID, Core Bitmaps and Event Indexes. Note that the schemes for Core ID and Core Bitmaps are identical to those for the FDC.

2.3.1 Core ID

We use the term Core ID to refer to a number that describes either a protocol core or an interface core in the *system*.

Cores are allocated to a cluster. There are three clusters in total, with each cluster containing five cores. Figure 6 illustrates the format that is used to create a Core ID. The Core ID is basically constructed from two

bits of Cluster Number, followed by three bits of Core Number. This Core ID format is the format used on the Message Bus. Note that for three clusters with five cores per cluster, the valid Core ID values are 0, 1, 2, 3, 4, 8, 9, 10, 11, 12, 16, 17, 18, 19, 20. Specifically, the Core ID values 5, 6, 7, 13, 14 and 15 are not valid.

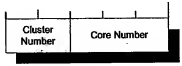

Figure 6: Core ID Format

2.3.2 Core Index

A Core Index is an encoding of the Core ID, as shown in Table 3. A Core Index provides a number in the range 0 through 14 with no holes, unlike a Core ID that has holes at 5, 6, 7, 13, 14, and 15.

Core Index	Core ID
0	0
1	1
2	2
3	3
4	4
5	8
6	9
7	10
8	11
9	12
10	16
11	17
12	18
13	19
14	20

Table 3: Core Index to Core ID Mapping

2.3.3 Core Bitmaps

Through out the Dispatcher we often need to keep a bitmap of cores. The question is in this bitmap what Core ID does bit i represent? One obvious choice is that bit i represents Core ID i. The problem with this is that not all Core ID's are valid, so the bitmap would be larger than it really needs to be. Since bitmap width is important, the Dispatcher uses another encoding in its Core Bitmaps. To solve this the Dispatcher uses a bitmap such that bit i represents Core Index i. Table 3 can then be used to convert this Core Index into a Core ID.

2.3.4 Event Index

We use the term Event Index to refer to a position in a Processor Cores Event Queue. Events are a fixed maximum size of 256 bytes, and there are a maximum of eight such events in an Event Queue. Event Indexes refer to 128 byte chunks of data within the area of memory reserved for events. Since events are a fixed size of 256 bytes, odd valued event indexes are never used.

2.4 Flow Director CAM Responses

2.4.1 Out of Resource Response

The Dispatcher interacts with the Flow Director CAM in order to determine where to forward events. In some cases the FDC can respond indicating that the FDC command was not processed due to lack of resources. The FDC can run out of the following resources:

- No space left in the CAM.
- No processor cores available to allocate.
- No workspace IDs available to allocate.
- No event indexes available to allocate.

The FDC indicates that it has run out of resources by setting the flags field of the response to 5'b11110.

The Dispatcher should repeatedly retry the FDC command until it succeeds. Note that it must attempt to service other queues first, before returning to the queue that caused the FDC *no resources* response.

2.4.2 Error Response

Hard errors occur due to bugs in software, or errors in the design. Hard errors are raised by the FDC when it encounters a command that given the state of the FDC entry, it does not believe to be valid. Examples of such hard errors are:

- Receiving a RMFIDX (remove FDC entry) when the FDC entry is in the *CHECKED-IN* state. *CHECKED-IN* means the entry does not exist, so it is not valid to remove it.
- Receiving a CRTIMER (create timer entry) when the FDC entry is in the *UPDATE* state. The LUC should not issue timer expirations for flows that are checked out, so this is not valid.

Hard errors are indicated by the FDC setting the flags field of the response to 5'b11111.

At first glance it would seem that if the Dispatcher encounters a hard FDC error then it should drop the event. However, due to the nature of the FDC there may be some cases where due to lack of analysis, commands are valid in states that we never expected them to be. The system would function perfectly if such FDC commands were retried at a later date. For example, we may determine that in fact a CRTIMER command is valid in the UPDATE state due to some very small window. For this reason if the Dispatcher encounters a hard FDC error we behave exactly the same as when the FDC runs out of resources, i.e. we leave the event at the front of its queue and attempt to forward it again at a later date[10].

2.5 Dispatcher and Scratchpad Interaction

2.5.1 Checksum, Byte Length and Error Information

For events received from either IPU, the following sequence of events occur:

- The packet processor receives the first 256 bytes of the packet or host message, parses it and forms a control event header as described in section 3.5.1. The packet processor calculates a 1's compliment checksum for the TCP portion of the packet, including the data up to the 256th byte. See RFC 1071 for information on a 1's compliment checksum.
- The packet processor may decide to issue a release to the Scratchpad. If it does this then it marks a flag in the control event to indicate that it has done so. An example would be when a TCP packet is received that contains no data.

[10] A bit is set in the status register so that we know this occurred.

- At the same time the IPU sends the whole of the packet to the Scratchpad. The Scratchpad will generate a 1's compliment checksum starting at the 257th byte. If the packet is less than 256 bytes then the scratchpad's checksum value is zero.
- For packets larger than 256 bytes, only the scratchpad knows the total byte length.
- If a DIP-4 error occurs on the SPI-4 interface then this is signaled to the scratchpad at the end of the data transfer.

You can now see that the packet processor has computed some of the 1's compliment checksum, and the scratchpad has the remainder of it. It is the task of the Dispatcher to combine these two checksums. Similarly, the scratchpad must fill in the total byte length and indicates if a DIP-4 error occurred. The Dispatcher does this as follows:

1. When the Dispatcher receives an event it examines the SP_DATA bit to determine if a scratchpad location is in use.
2. If a scratchpad location is in use then the Dispatcher issues a query to the Scratchpad, requesting information about the checksum, total byte length, and any error information. It uses the Frame ID of the input event to request this information. At some point, the scratchpad will respond.
3. If a scratchpad location is not in use then the Dispatcher does not query the scratchpad.

Only the IPU0 and IPU1 interfaces require this interaction with the scratchpad. There are also two dedicated buses on the scratchpad for performing these requests. One such bus can be used for IPU0, the other for IPU1. These buses are labeled as SP_BUS0 and SP_BUS1 on Figure 1.

For this to work properly the scratchpad stores a READY bit with each Frame ID. This bit is set after a complete transfer has occurred from the IPU. The scratchpad clears this bit when the Dispatcher requests checksum information OR when the packet processor releases the Frame ID. When the Dispatcher requests checksum information, the scratchpad does not respond until the READY bit is set. This ensures that we do not use the checksum, byte length or error information from the pervious packet that occupied this Frame ID.

2.5.1.1 Checksum Alignment Issues

As described above, the task of computing the 1's compliment checksum of a packet is split between the packet processor and the scratchpad. The packet processor computes the checksum over the first 256 bytes, the scratchpad over the rest. Note that the 1's compliment checksum is a 16-bit checksum, and as such operates on 16-bit values.

Splitting the checksum between the packet processor and the scratchpad works well when the split of data is on a 16-bit boundary. This situation is shown in Figure 7. If we use [a,b] to represent the value a * 256 + b, and we use "+" to represent 1's compliment addition, then with regards to Figure 7 the packet processor computes [A,B] + ... + [Q,R] + [S,T]. The scratchpad computes [U,V] + [W,X] + ... + [Y,Z]. The Dispatcher then combines these to form the complete checksum of [A,B] + .. + [Q,R] + [S,T] + [U,V] + [W,X] + ... + [Y,Z].

Figure 7: Checksum with Scratchpad Data 16-bit Aligned

Now consider the case where the data is received such that the split between the packet processor and the scratchpad does not occur on a 16-bit boundary. Figure 8 illustrates this situation. In this case the packet processor will compute [A,B] + ... + [P,Q] + [R,S] + [T,0] and the scratchpad will compute [U,V] + [W,X] + ... + [Y,Z]. If these are combined without any additional operations then the checksum would be [A,B] + ... + [P,Q] + [R,S] + [T,0] + [U,V] + [W,X] + ... + [Y,Z]. This is not the expected checksum of [A,B] + ... + [P,Q] + [R,S] + [T,U] + [V,W] + [X,Y] + ... + [Z,0].

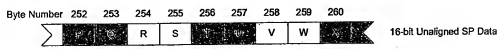

Figure 8: Checksum with Scratchpad Data 16-bit Unaligned

Note that the following two properties are true of a 1's compliment checksum (see RFC 1071 for further details):

if [a,b] = [U,V] + [W,X] + ... + [Y,Z] [Rule 1, Byte Order Independence]
then [0,U] + [V,W] + ... + [Z,0] = [b,a]

[a,0] + [0,b] = [a,b] [Rule 2, Commutative and Associative]

Suppose the scratchpad computes the checksum and its value is [a,b]. Given the above rules, we know that:

[A,B] + ... + [P,Q] + [R,S] + [T,0] + [b,a] = [A,B] + ... + [P,Q] + [R,S] + [T,0] + [0,U] + [V,W] + ... + [Z,0]
= [A,B] + ... + [P,Q] + [R,S] + [T,U] + [V,W] + ... + [Z,0]

Therefore, if the split between the packet processor and the scratchpad is not on a 16-bit boundary, then the checksum from the scratchpad should be byte-swapped before it is combined with the packet processors checksum.

The next problem is how does the Dispatcher know whether the packet processor / scratchpad split was on a 16-bit boundary? The TCP checksum starts at the TCP pseudo header and continues through the TCP header, TCP options and TCP data. The TCP pseudo header is a multiple of 16-bits in length, as are the TCP header and TCP options. Therefore, if the TCP checksum started on a 16-bit boundary, then the TCP Data starts on a 16-bit boundary and vice-versa. A pointer to the TCP Data is provided in the Control Event to the Dispatcher, so it can determine the least significant bit of that field to determine whether to swap the scratchpads checksum.

2.5.1.2 Checksum Length Issues

The Dispatcher performs a checksum on **all** data after the first 256-bytes. This can present problems with packets that contain padding data. For example, consider a TCP packet where the actual length of the packet is one byte beyond where the TCP payload ends. Although it would be unusual to have a packet that is larger than 256-bytes and contains such pad, it is still possible[11]. If such a packet were received then the Dispatcher would checksum all of the data, including the one byte at the end of the TCP payload. Without correction, this would make it look like the packet had a bad TCP checksum.

The software must resolve this checksum length issue. The software is informed of the total packet length via the *Scratch Data Length* field of Figure 23. The software can use this value, along with the indication of the payload length (i.e. IP length for TCP checksums) to determine if any padding data has been checksummed. The software must then undo any padding that is part of the checksum. Note that due to the quantities of data involved, it is preferential to undo the data that was checksummed rather than redo the whole checksum.

2.5.1.3 Race Conditions

Since the packet processor, IPU, scratchpad and Dispatcher all operate in parallel, there was concern over possible race conditions in the checksum, byte length and error information handling. These race-conditions occur when the same Frame ID gets re-used before the Dispatcher has had a chance to gather the checksum information about it. Although these were initial concerns, the Dispatcher / Packet Processor / Scratchpad interaction was designed such that this can never occur. The reasoning behind this is described below.

[11] A bug in the peer's packet driver could cause this, or perhaps some intermediate device added extra pad to align the packet length on a nice boundary.

A Frame ID cannot be re-used until the scratchpad receives a release for that Frame ID. That release will either come from the packet processor, or it will come from an element *after* the Dispatcher. The Dispatcher itself cannot issue releases.

If the Packet Processor marks an event as having the scratchpad released, then Dispatcher does not query the scratchpad. Since it does not query it, it does not matter if the release has or has not been processed by the scratchpad. When the release is processed, that Frame ID is available again for re-use.

If the Packet Processor marks an event as having a scratchpad location, then the Dispatcher must query the scratchpad and then forward the event. We know there is no danger of the Dispatcher getting stale checksum information, since that Frame ID will not be used again by the scratchpad until the element down the line releases it.

The purpose of the READY bit is to hold off the Dispatcher until the checksum is ready. Note that the READY bit is always cleared before the Frame ID is re-used. Either the packet processor clears it on the release, or the Dispatcher clears it when it reads the checksum information. That bit is never set again until the scratchpad has received another complete packet. For this reason the Dispatcher can only query the scratchpad once for checksum information. This is important to remember when processing the multicast stateless events described in section 1.4.1.2.3.

2.5.2 Reference Count Increments

As described in section 1.4.1.2.3, the Dispatcher can send the same event to multiple processor cores. This event can have a scratchpad location associated with it, i.e. it has an assigned Frame ID. Each Frame ID in the scratchpad keeps a *reference count* that it uses to determine when it can be released. By default this reference count is set to one when the Frame ID is created. The typical scenario is that the last person who uses the scratchpad location issues a release command. A release command decrements the reference count, and when the reference count reaches zero the scratchpad location is released.

The processor cores have the ability to increment this reference count – they would do this when more than one source is using the scratchpad. For example, if a processor core wishes to send a Frame ID to the OPU, and then send a message to another processor core that will also reference that Frame ID, then it must first increment the reference count. The reason for this is that both the other processor core and the OPU will issue a release command when they are done.

This brings up an interesting issue with events that have a scratchpad location associated with them, but who are sent to multiple processor cores, i.e. repeated unicast stateless events. Each processor core must issue a release after it has finished processing that event. How do we ensure that the scratchpad location is not released until the last core has finished using it?

At first it would seem that this problem could be solved by software. However, by further examination it becomes apparent that no processor core can tell when the other cores have finished, so no processor core can issue the release. Instead we must make the Dispatcher increment the reference count by N – 1, where N is the number of processor cores that this multicast event will be sent to. After the last processor core issues the release, the reference count will reach zero, and the scratchpad location will be released. This reference count increment is dealt with in the flow chart of Figure 18.

To perform the reference count increase the Dispatcher uses the Scratchpad Reference Count Bus, illustrated in Figure 1 as SP_CNT_BUS. Note that the Scratchpad reference count is incremented regardless of whether this event came from IPU0, IPU1, or the SMC. This is unlike the checksum information that is only communicated for events from IPU0 or IPU1.

2.6 Flow Charts

2.6.1 Flow Chart Usage

This document uses flow charts to describe the logic that the Dispatcher follows. These flow charts are inherently sequential as a means of describing logic. This should not stop the real implementation from using pipelining or parallelism in order to accelerate performance, as long as the logic of the flow charts is not broken. For example, assuming access to shared resources (FDC, output queues, counters) is correctly handled, each input queue could run its own state machine that follows the logic described in these flow charts. Another example is to do something useful while waiting for FDC responses. These are left as decisions for the hardware designer to make and validate.

2.6.2 Query Scratchpad Subroutine

The *Query Scratchpad* subroutine of Figure 9 is used to obtain checksum, byte length and error information from the scratchpad. The details and reasons for this interaction are described in section 2.5.

The first thing that we check in this subroutine is whether the input event came from IPU0 or IPU1, and whether the SP_DATA flag is set in the input event. If any of these conditions are false then we do not query the scratchpad for checksum information, and the *Running Checksum* and *Scratch Data Length* fields of the input event are not modified.

If the event came from IPU0 or IPU1, and the SP_DATA flag is set, then we issue a query to the scratchpad. For this query we supply the Frame ID of the input event. We then wait for a response from the scratchpad. The amount of time we have to wait depends on how busy the scratchpad is, and whether it has completely received that Frame ID.

When the scratchpad responds it supplies a byte length, checksum and error bit. For the reasons described in section 2.5.1.1, we must examine the least significant bit of the *Payload Scratch Offset* field of the input event. If that bit is set then we must byte-swap the checksum that is returned from the scratchpad. We then perform a 1's compliment addition to combine the Running Checksum of the input event with the possibly byte-swapped checksum from the scratchpad. For more information on the 1's compliment checksum the reader is directed to [RFC 1071].

Finally we write the *Running Checksum* back to the input event, write the byte length back as the *Scratch Data Length*, and combine the error bit from the scratchpad with the existing *DERR* bit. It is important that these are written back to the input event, since that is where we store this information for repeated unicast events.

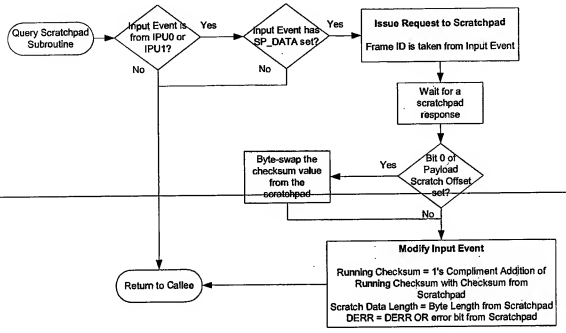

Figure 9: Query Scratchpad Subroutine

2.6.3 Dispatcher Start

Figure 10 shows the top-level flow chart for the Dispatcher. Here we simply service each input queue of the Dispatcher in a round robin fashion. In this flow chart "Event is present" requires us to see if an element is in a queue. Note that this does not mean that the element can be removed from the queue without first following the *Process Event* or *Process Timer Event* flow chart, since not all events can be serviced immediately.

First we check for a Timer Event. If one is found then we use the *Process Timer Event* flow chart of Figure 14 to process it.

Next we check to see if we are allowed to service IPU0 or IPU1 events. Such events cannot be serviced if the SMC_DISP_Almost_Full signal from the SMC is set. This signal is used by the SMC to indicate that it is starting to queue events to DDR memory. To prevent extra events entering the system, the Dispatcher effectively disables the IPU0 and IPU1 interfaces while this signal is set[12]. This is the default behaviour of the Dispatcher, but that can be changed via the DIS_SMC_DISP_BP bit in the CONTROL register.

Assuming that SMC_DISP_Almost_Full signal is not set, we next check for an event from the IPU0 or IPU1 bus. If such an event is present then we check the DERR bit. If this bit is set then the Dispatcher is being asked to drop the event and we continue with the *Drop Event* flow chart of Figure 16. If the DERR bit is not set then we continue with normal event processing using the *Process Event* flow chart of Figure 11. Note that the DERR bit is only examined for events from the IPU0 or IPU1 bus.

Then we check for SMC events, using the *Process Event* flow chart of Figure 11 if one is found.

[12] This is part of a deadlock avoidance algorithm. See the *Deadlock Analysis and Avoidance* document for further details, specifically the section on *Limited Dispatcher Feedback Queue(s) Block Done Events*.

Next we check to see if the event is a Done Event. Done Events are a special type of event that may cause us to issue an update message to the LUC. These are processed by the *Process Protocol Done Event* flow chart of Figure 15.

Note that in all cases, when an event is found, we increment the received event counter register (RCV_EVNT_CNT), a useful statistic for debugging the Dispatcher.

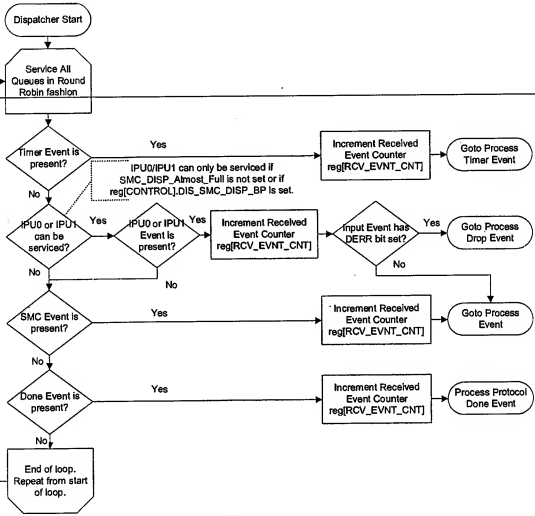

Figure 10: Dispatcher Start Flow Chart

2.6.4 Process Event

Figure 11 illustrates the actions that occur when we have determined that the Dispatcher must process an event.

First we must determine how the event should be forwarded. The first check we make is whether the most significant bit (bit 5) of the Event Type is set. If it is set then we know that this is a stateless unicast forward (see section 1.4.4). If the most significant bit is not set then we use the lower 5 bits of event type (ETYPE) to

access the event parameters (EPARMS) register set and examine the forwarding mode (FW_MODE). Based on this forwarding mode to continue with the appropriate flow chart.

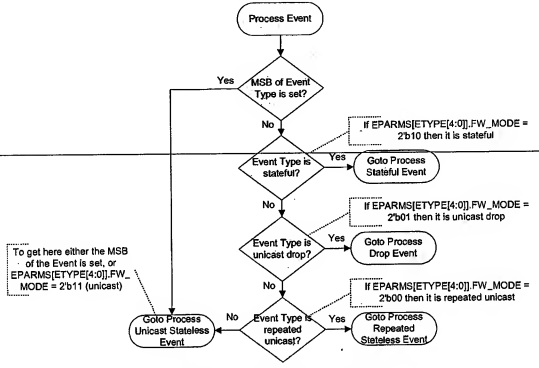

Figure 11: Process Event Flow Chart

2.6.5 Process Stateful Event

Figure 12 illustrates the actions that are taken when the Dispatcher receives an event that requires the LUC, i.e. is stateful. First we issue a Lookup with Flow Key[13] and Create command (LFKCREATE) to the Flow Director CAM (FDC). This command causes the FDC to lookup the Flow Key in its CAM. If no entry is found then one is created. If an entry is found then the response from the FDC will indicate the *new* state of that entry. For further details on the FDC response see "Flow Director CAM (FDC) High Level Design".

We then make a series of comparisons against the response from the FDC. If the FDC response indicates an error then we log it and then return to the Dispatcher start (see section 2.4 for further explanation). If the FDC response indicates that there is no space in the FDC[14] then we leave the current event as outstanding on the input queue, and return to the main Dispatcher loop. The reasoning here is that we will service the event at a later date, hopefully when there is more room in the FDC.

Assuming that we did not receive an error response from the FDC, and that we were not told that the FDC is full, we can start to examine the Flags field of the FDC response to see what state the entry is now in. The first two states that we look for are:

- **Entry is in the *DELETE* state.** In this scenario a tear down request has been issued to the LUC, but then another event arrives for that flow. See section 2.6.11 for more details.

[13] Packet, host and timer events all have an associated flow key.
[14] This lack of space in the FDC could be because the FDC CAM is full, or because there is no space in the Processor Cores event queues and workspace IDs.

- **Entry is in the *TIMER* state.** This is the case where the TCU has created an FDC entry and issued a timer event, but the Dispatcher has not yet got round to servicing it. In this case we **MUST** service the timer event before any packet or host events that may have arrived. Timer event servicing is described in more detail in section 2.6.6. Note that at the start of the *Process Stateful Event* flow chart we were processing an event from one of the IPU0, IPU1, or SMC queues. However, when we goto the *Process Timer Event* flow chart we are processing the event on the **Timer Queue**, and that is the event that **must** be removed when *Process Timer Event* calls the *Remove Input Event* flow chart.

If the FDC entry is not in the *DELETE* or *TIMER* states, then it must be in either the *RECEIVED* or *PENDING* state. We then continue with the *Issue LUC Command Flow Chart* of Figure 13.

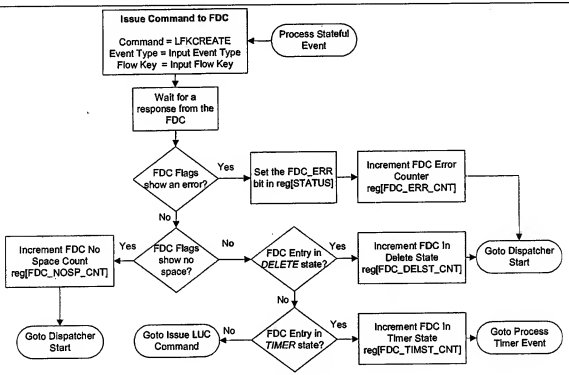

Figure 12: Process Stateful Event

2.6.5.1 *Processing Entries Found in the* DELETE *State*

If an FDC entry is found in the *DELETE* state then we must be in the process of tearing down the flow. For example, this situation could occur when:

1. The network duplicates frames, e.g. the ACK that is received in the LAST_ACK state gets duplicated.
2. A client's TCP/IP stack allows a TCP connection to live past the 2MSL (Maximum Segment Lifetime) value.

As shown in Figure 12, in this scenario we do not attempt to undo the tear down operation that has already been issued, but instead we wait until the tear down has completed. When the LUC completes the tear down the FDC entry will be removed. When the Dispatcher then re-issues the LFKCREATE command a new FDC entry will be created and a command will be issued to the LUC to lookup the state for this flow. That flow has just been deleted, so no state will be found and the processor core will get a "could not find state" response[15]. The Processor Core can then determine the correct response, e.g. send a reset.

Note that head of line blocking will occur until the LUC removes the entry from the FDC, i.e. head of line blocking occurs while the FDC entry is in the DELETE state. See section 2.7 on blocking events for more details.

2.6.6 Issue LUC Command Processing

In Figure 13 we determine what, if any, LUC command should be issued. From Figure 12 we already know that the FDC entry is in the *RECEIVED* or *PENDING* state. The first thing we do is to check to see if the FDC entry has just been created, i.e. whether the FDC entry has **just** transitioned into the *RECEIVED* state from the *CHECKED IN* state[16]. If the FDC entry is not a new entry then a lookup request has already been issued, and another **MUST NOT** be sent. In this case we simply create a Processor Core event using the information that was found in the FDC.

If the FDC does show that this is a new entry then we must look at the *create* bit in the event. This bit is used to determine whether the LUC should create an entry if the flow key cannot be found. Note that we also use the EPARMS[ETYPE].LUCCM1 and EPARMS[ETYPE].LUCCM2 register values so that we can program the commands that are issued to the LUC on a per event type basis. For example, In some circumstances we would want to issue a lookup with listen and create, and in other circumstances we just want to do a plain lookup with create.

If the create bit is not set then we must determine if this event type uses the Flow Key for the lookup[17], or whether it uses the Socket ID. An example of using the Flow Key is a TCP event: the only information we have available is the Flow Key, so this must be used for the lookup. An example of using the Socket ID is with host events: the application records the Socket ID that was assigned when the socket was created and it always passes this information along with the Flow Key on all host events[18]. Since using the Socket ID is more efficient for the LUC, we use that method of lookup wherever possible.

We then call the *Query Scratchpad* subroutine of Figure 9. This subroutine may modify the input event depending on whether it came from IPU0 or IPU1 and has the SP_DATA bit set. For more details the reader is directed to section 2.6.2.

We then modify the input event based upon the FDC response. Based upon the Event Mode of the FDC response, we then forward the event to either the Protocol Core or the Interface Core using the supplied Event Index.

[15] If the correct flags were set in the TCP header then the Packet Processor may have set the create bit, in which case a new flow will be created. The Processor Core will deal with this case as if it were a new flow, which is the correct behaviour.
[16] The FDC response to an LFKCREATE command uses a single bit to indicate if this is a new entry. See the FDC HLD for further details.
[17] Note that we only use the lower 5 bits of the Event Type since we know this is a stateful event. See section 1.4.4 for further details.
[18] This is a requirement of the application interface.

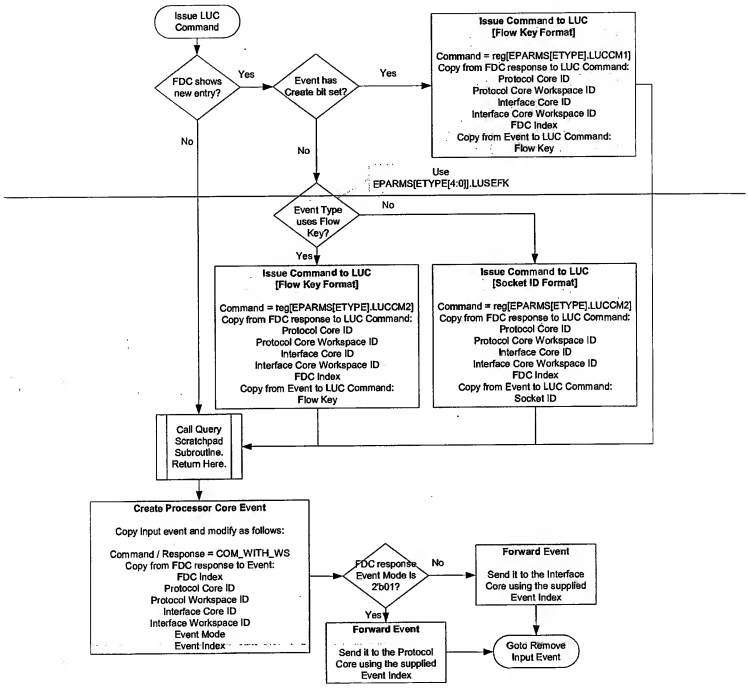

Figure 13: Issue LUC Command Flow Chart

2.6.7 Process Timer Event

The *Process Timer Event* flow chart is used when an entry is found in the FDC that is in the *TIMER* state, or when we see that the timer queue is not empty. Only the TCU can create *TIMER* entries in the FDC, and it

will do so when a timer expires. The purpose of creating such an entry is to ensure that if a timer expires while servicing a packet / host event[19], we always service the timer before servicing the packet / host event. The reason is that servicing a packet or host event may cause the timer to be cancelled, but by this point there is no way to back out the timer expiration. Note that once a flow is checked out from the LUC[20] no timer events will be issued for that particular flow until it is checked back in, at which point the LUC may cancel pending timers based upon the new timer values (as set by the Processor Cores).

The first task we must perform is to assign a Processor Core and workspace ID pair to this particular FDC entry: the TCU simply creates the FDC entry and does not assign any Processor Cores. To perform this assignment the SERVTIMER command is issued to the FDC. Note that for the SERVTIMER command we use the Event Type value from ETSERV field of the ECPARMS register. Note that this Event Type must be stateful, so the MSB must be zero (see section 1.4.4.1). For this reason we only store 5-bits of Event Type in ETSERV, and then we explicitly set the MSB to zero in the FDC SERVTIMER command.

We must then examine the response from the FDC to determine if the SERVTIMER command was successful. If the FDC indicates that an error occurred then we log that fact and continue with the main Dispatcher loop (see section 2.4 as to why the event is not dropped). If the FDC indicates that there was no space available then the timer event is left outstanding and we continue with the main Dispatcher loop. Just as in section 2.6.5, the hope is that eventually there will be resources available in the FDC so the timer event can be serviced at a later date: we just keep on trying.

If the FDC did not indicate an error, and there was space available, then the FDC entry must be in the *RECEIVED* state. We must now issue a lookup request to the LUC in order to get the flow state written into the Processor Cores workspace ID. Note that the LUC command that we use is programmable, but it is expected to be a *lookup with Socket ID (LSID)*[21]. We then move the timer event to the assigned Processor Core and event queue element, as indicated by the FDC response. Note that when creating the Processor Core event, we set the Socket ID to zero even though we know the correct value. Setting the Socket ID to zero is consistent with the operation of the Dispatcher for non-timer events, where it does not modify the Socket ID.

For Timer Events we do not query the Scratchpad for checksum, byte length or error bits. The reason is that timer events have no associated payload in the Scratchpad.

[19] The critical section here is between the Dispatcher servicing the packet / host event, and the LUC receiving the lookup command for that flow (checking the flow out).
[20] A flow is checked out in between the time the LUC processes the lookup request and receives an update or tear down request.
[21] Using a lookup with Socket ID rather than a lookup with Flow Key saves the LUC from having to do a hash list search.

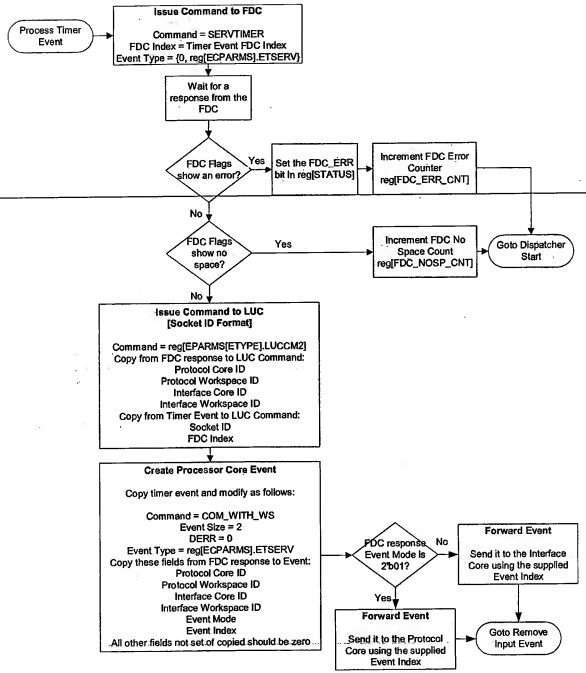

Figure 14: Process Timer Event Flow Chart

2.6.8 Process Protocol Done Event

The *Process Protocol Done Event* of Figure 15 is used when the Dispatcher receives a done event. The format of these done events is described in section 3.5.1 and Figure 23. Such done events use different event types to indicate whether this is a *Done and update flow* or *Done and tear down flow*.

The first check we make is whether the most significant bit (bit 5) of the Event Type is set. If it is set then this must be a stateless event type (see section 1.4.4). If the most significant bit is not set then we use the lower 5-bits of the Event Type to access the Event Parameter registers set.

If the event type of the done event indicates that it is stateless then we simply send the FDC a RELEVENT command, using the event index and Processor Core from the done event. An example of this would be when a Processor Core has finished processing an ARP frame: there would be no associated workspace ID but the Dispatcher still needs to free up the event queue resource in the FDC.

If the event type of the done event indicates that it is stateful then we check to see if the event type is a "Done and tear down flow". For a done event of the tear down type we send a TDFIDX command to the FDC, otherwise we send a UPFIDX command.

If a UPFIDX or TDFIDX command was sent then we must examine the response from the FDC. First we check if the FDC has indicated that an error occurred. If it has then we log that fact and then continue with the main Dispatcher loop. Note that we do not remove the input event from the queue if the FDC indicates an error (see section 2.4 for details).

The next check we must make is whether the FDC entry is in the *PENDING* state. If it is in the *PENDING* state then we must wait for the LUC to complete the previous update it has already been given[22]. If this is the case then we simply increment a counter and return to processing other Dispatcher queues. Note that the original done event is not removed from the input queue: the hope is that the next time we service this same event the FDC entry will no longer be in the *PENDING* state.

If the response indicates that the FDC entry is now in the *DELETE* state then we send a tear down command to the LUC[23] using a flow key format. If the response indicates that the FDC entry is now in the *UPDATE* state then we send an update command to the LUC using a socket ID format[24]. If the FDC entry is neither in the *DELETE* or *UPDATE* state then we must be in the *RECEIVED* state and no interaction with the LUC is required.

With reference to the state transition diagram of an FDC entry, it is not possible to issue a TDFIDX command and then find the entry in the *UPDATE* state. Similarly, it is not possible to issue an UPFIDX command and then find the entry in the *DELETE* state. The logic in flow chart of Figure 15 could therefore be changed to reduce the number of comparisons, but in the interests of simplicity this document does not do so.

[22] In order for the FDC entry to be in the *PENDING* state it must have already been issued an UPFIDX command. What has happened is that in the time it has taken the LUC to update the flow's state another event has arrived for the same flow. A Processor Core has processed this new event and another done message has been issued. All this occurred before the LUC could finish the original UPFIDX command.
[23] According to the state machine of each entry in the FDC, the UPFIDX and TDFIDX commands will respond with an error in all cases except when the FDC entries original state is *RECEIVED*. We therefore know that only one update or tear down request will be issue to the LUC.
[24] Using the Socket ID formats for updates is more efficient since it saves the LUC having to traverse it's hash tables.

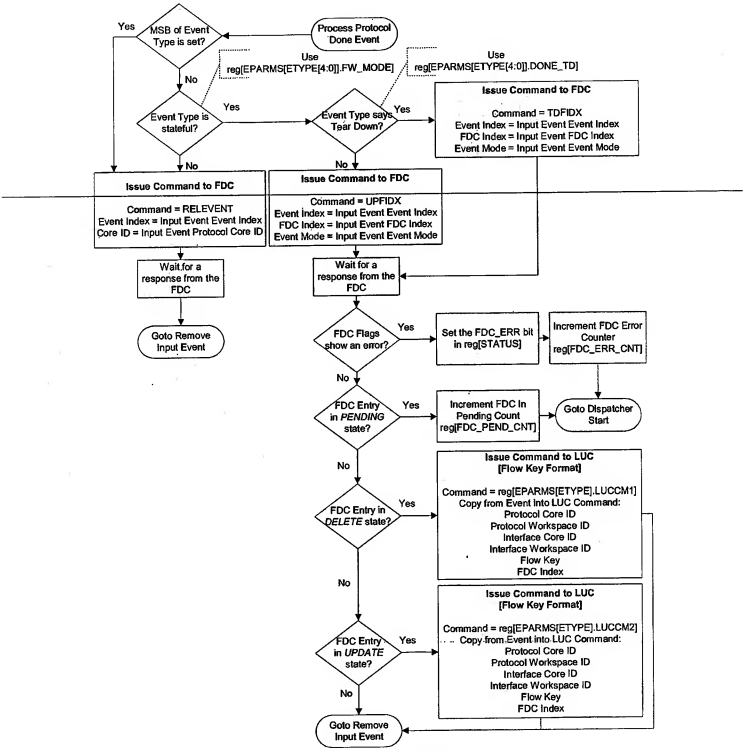

Figure 15: Process Protocol Done Flow Chart

2.6.9 Process Drop Event

Figure 16 illustrates the steps that are taken when the Dispatcher wants to drop an event. The steps are very simple. First we issue a command to the FDC that requests that a processor core and event index be allocated for this event type. Based on the results of that we either forward the event, or increment a counter to say that the FDC response indicated no space.

If there was no space in the FDC then note that this event is left at the head of its queue. It will be examined again the next time the Dispatcher services that queue. This is the expected behaviour, i.e. we indefinitely retry to send the event.

We then call the *Query Scratchpad* subroutine of Figure 9. This subroutine may modify the input event depending on whether it came from IPU0 or IPU1 and has the SP_DATA bit set. For more details the reader is directed to section 2.6.2.

If we do forward the event then note how we mark it as a drop event by setting the DERR flag. See Figure 23 for the location of the DERR flag.

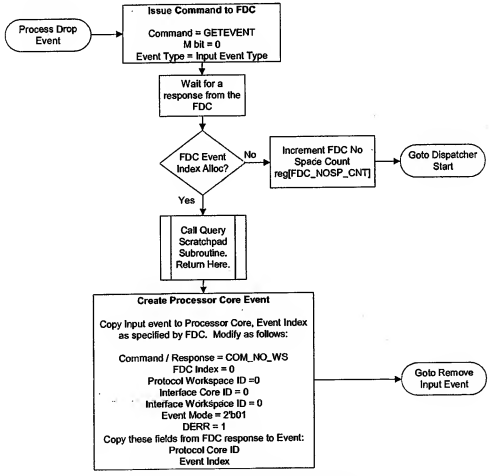

Figure 16: Process Drop Event Flow Chart

2.6.10 Process Unicast Stateless Event

Figure 17 illustrates the steps that are taken when the Dispatcher wants to forward a unicast stateless event. These steps are exactly the same as those described in section 2.6.9 above except this time the DERR flag is not set.

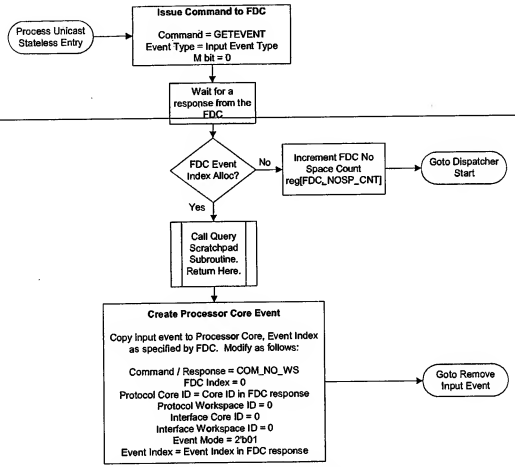

Figure 17: Process Unicast Stateless Event Flow Chart

2.6.11 Process Repeated Unicast Stateless Event

The *Process Stateless Event Flow Chart* of Figure 18 illustrates the steps that must be performed when processing a repeated unicast stateless event. With reference to the *Process Event Flow Chart* of Figure 11, we know that if we are performing repeated unicast stateless event processing then the most significant bit of the Event Type is zero. Therefore, when accessing the Event Parameter registers we only use the lower 5-bits of the Event Type. If the most significant bit of the Event Type is only set for unicast stateless event processing (see section 1.4.4).

As noted in section 1.4.1.2, stateless events can be multicast to one or more Processor Cores. In order to do this the Dispatcher keeps a bitmap of *cores that have been sent* to, with one bitmap per input queue. In Figure 18 we call this set of registers CUR_CREPEAT[Q], where Q represents the particular input queue.

The first step we take in stateless event processing is to examine whether CUR_CREPEAT[Q] is zero. If it is zero then we must be processing a new stateless event multicast. Since we are starting a new multicast, we examine the SP_DATA bit to determine if we must increment the reference count on the Scratchpad location according to section 2.5.2. If a Scratchpad location has been allocated, then SP_DATA would have been set to one by the Packet Processor, and the Dispatcher must issue an increment by N command, where N is the number of bits in the CREPEAT bitmap[25]. Note that this increment is done for events from the IPU0, IPU1, and SMC queues. After that command has completed, we then call the *Query Scratchpad* subroutine of Figure 9. This subroutine may further adjust the input event if it came from IPU0 or IPU1. For further details the reader is directed to section 2.6.2. Due to the position of this *Query Scratchpad* subroutine call, it will only be called once each time we start a fresh repeated unicast. This is important, since it is not valid to query the Scratchpad for every single iteration of the repeated unicast (see section 2.5.1). Finally, we initialise CUR_CREPEAT[Q] to the value of the CREPEAT register for this event type (ETYPE).

If the CUR_CREPEAT[Q] register was not zero then we must be in the middle of a previous multicast, and we are re-entering this flow chart to see if we can send any more events out.

We then enter a loop that is repeated until CUR_CREPEAT[Q] is zero, or until the FDC reports that there is no space left. This loop sends a GETEVENT command to the FDC, using the current Core Bitmap of Processor Cores that need to be sent to. It should be noted that by setting the M bit of the GETEVENT command we can specify a Core Bitmap of Processor Cores that should be considered. The FDC then assigns an event queue element based on which, if any, of those Processor Cores have available space.

The FDC response to the GETEVENT command is then examined to see if a Processor Core event queue element was assigned. If a Processor Core could not be assigned then we simply return to the Dispatcher main loop, leaving the value of CUR_CREPEAT[Q] as non-zero. Since we have not removed the input event from the input queue, we will always come back and service this stateless event again.

Assuming that space was available in a Processor Core, we now move the stateless event onto the Processor Core that was assigned. Note that, by examination of the command / response field of the event, the Processor Core must realise that this event type does not have any associated workspace ID.

Finally we remove the Processor Core that was serviced from the CUR_CREPEAT[Q] bitmap, and continue with the loop in the flow chart. This will cause us to keep sending events to Processor Cores until CUR_CREPEAT[Q] is zero, or until the FDC response indicates that an event queue element could not be assigned.

[25] Note that if only one bit is set in the CREPEAT bitmap, then N = 1 and an increment of zero will be sent to the Scratchpad. This is valid, i.e. the Scratchpad can accept an increment of zero, and will ignore it. It is an invalid configuration to have zero bits set in the CREPEAT bitmap.

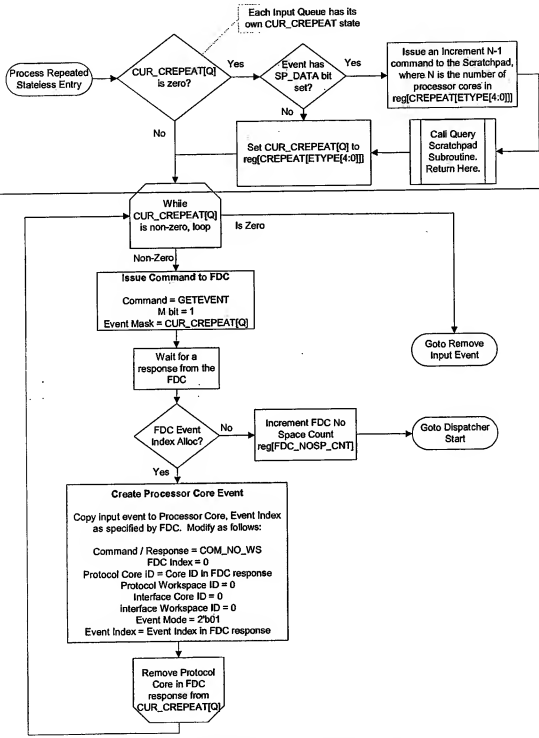

Figure 18: Process Repeated Unicast Stateless Event Flow Chart

2.6.12 Remove Input Event

The *Remove Input Event* flow chart of Figure 19 is very simple: it removes an input event from an input queue and increments a counter to indicate that an event has been processed. The only reason for having this case explicitly described is that some processing will not remove the input event from the input queue, e.g. when processing cannot continue because the FDC is full. We distinguish between this case and the case when the event has finished being processed by explicitly calling the *Remove Input Event* flow chart.

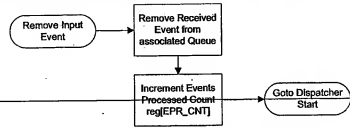

Figure 19: Remove Input Event Flow Chart

2.7 *Blocking Events*

The design of the Dispatcher described in this document will cause the Dispatcher to block on certain events, these being:

1. FDC is full.
2. Processor Core and workspace ID could not be allocated by the FDC. The event type defines which Processor Cores can process an event, so there could be a case where some Processor Cores are available but they are not suitable for the input event type. Also note that if the FDC is operating in dual core more, then both a Processor Core and Interface Core must be available, and each of them must have an available workspace ID.
3. A protocol done message is received, but the FDC entry is found to be in the *PENDING* state. We must wait until the LUC has finished updating the entry before the protocol done message can be completed.
4. An event arrives but the FDC entry is found to be in the DELETE state. We must wait until the LUC has finished removing the entry before the event can be forwarded to a Processor Core. This is an unusual event in that an event was received for a flow that is being torn down. Note that this scenario is only blocking for the first event, after which events will flow freely until the next tear down is issued.
5. A stateless event needs to be multicast to multiple cores, but none of the cores left in the multicast are available.

In these cases it is possible that events behind the blocking event could be serviced, but they cannot be due to head of line blocking. However, the Dispatcher has multiple input queues so that hope is that while one queue may be blocked, other queues can be serviced. In the absolute worst case all queues will be blocked and the Dispatcher must perform a busy wait[26].

2.7.1 Denial of Service (DoS) Attacks

Given the fact that the Dispatcher does have blocking events, it could be possible to create a stream of traffic that reduces the performance of the Astute Content Processor (ACP), i.e. a malicious user could deny service to other users. Possible denial of service attacks related to the Dispatcher include:

1. Sending back-to-back packets that cause a repeated unicast to be sent to all Processor Cores.
2. Sending back-to-back TCP packets. All these packets will be directed to a single TCP core that will not be capable of processing at wire speed. This will eventually cause packets to be dropped, including packets that are not part of the back-to-back stream.

[26] It is not possible for the Dispatcher to enter deadlock. If all queues were blocked then the LUC must be in the process of updating an entry. When it finishes we should be able to move an event on the done queue.

3. Directing repeated link layer multicasts / broadcasts to the ACP, e.g. ARP requests. Either these messages will be multicast to a group of Processor Cores, or a single Processor Core will be assigned to process them. In either case those messages will overwhelm the available processing resources, causing packets to be dropped.
4. Sending back-to-back packets that cause all workspaces on all Processor Cores to be allocated. In this scenario any timer events that are in the Timer Queue will not be serviced since the SERVTIMER command will get a *No Space* response from the FDC due to a lack of Event Indexes or Workspace IDs[27]. Even if a Done Event does de-allocate an Event Index or Workspace ID, then the next queue to be serviced will be an IPU0/1 or SMC queue, which could re-allocate that Event Index/Workspace ID to another flow. By the timer the Timer Queue is serviced again there will be no Event Indexes or Workspace IDs available. The Timer Queue is effectively being starved, and as such the Processor Cores will not process any timer expirations. If this were a serious problem then one way to combat it is to assign a unique Event Type to timer expirations, and to dedicate a Processor Core to timer event processing and nothing else. That way we know that the timer expirations will get serviced, since the back-to-back packets cannot interfere with the Processor Core dedicated to timers.

The prevention of such denial of service attacks relies on both security by the customer (no direct access to the LAN so link layer broadcast attacks cannot occur) and careful selection and installation of firewalls (to filter e.g. excessive ICMP destination unreachable messages, excessive TCP frames from a single customer).

The architecture and software of the ACP must also reduce the likely hood, and effect, of these DoS attacks. It is believed that the Dispatcher architecture meets these guidelines.

2.8 Expected Performance

In this section we give an insight into the expected performance of this device. Note that this quick calculation is meant to guide implementation options, and is not meant to be a precise metric which can be used as a pass / fail test of the Dispatcher. The true required performance will be guided by the various simulations.

We use multiple methods for estimating the Dispatcher expected performance: the method that produces the highest performance requirement dictates the performance level of the Dispatcher. Also note that these performance requirements are not all required at the same time. For example, the connections per second metric and the timer metric are disjoint: we use the connections per second metric or the time metric but not both at the same time.

2.8.1 Dispatcher Performance Based on Connections Per Second Metric

This metric is based on a number of TCP connections being established and torn down with very little data transfer in between, e.g. lots of HTTP transactions. Processing an event requires the following steps:

1. The Dispatcher receives the event and due to the event type it arrives at the *Requires LUC Flow Chart* of Figure 12. We assume that this event requires the LUC and that the entry does not exist in the FDC. We therefore perform these actions:
   a. Issue an LFKCREATE command to the FDC and wait for the response[28].
   b. We find that the FDC entry is new, so we issue a LUC command.
   c. An event is created in the Processor Cores event queue.
   d. The input event is de-queued.

[27] Due to the number of entries in the FDC we know there will be *some* entries in the FDC where the CRTIMER command was successful, i.e. lack of CAM space cannot cause this problem. See the *FDC HLD* for details.

[28] The FDC HLD defines performance in terms of 500K connections per second. This assumes that the FDC and Dispatcher are perfectly overlapped, which may not be true. Tuning the FDC response time will require co-operation from the engineers implementing the FDC and Dispatcher.

2. When the Processor Core has finished processing the above event, it issues a "Done Event" to the Dispatcher. This is processed using the *Process Protocol Done Message Flow Chart* of Figure 15. The actions required are:
   a. Issue an UPFIDX command to the FDC and wait for the response.
   b. We find that the FDC entry is in the UPDATE state, so we issue a LUC command.
   c. The input event is de-queued.

The above commands are the minimal that are needed for completely processing a packet or host event. For now we do not consider timers. Also note that on the last frame of a flow the tear down with FDC index (TDFIDX) command will be issued rather than a UPFIDX.

Let us assume that the above steps are issued for each host or packet event in a flow. Let us then state that when measuring connections per second, there will be eight such host or packet events per flow[29]. Therefore, if we require 500,000 connections per second, then we expect 4,000,000 host or packet events per second, i.e. we expect steps 1.a through 2.c to be executed 4,000,000 times per second.

2.8.2 Dispatcher Performance Based on Bulk Transfer Metric

This metric is based upon a small number of TCP connections that are established at start of day, and then used to transfer large amounts of data.

The current architecture uses a 10Gbit full duplex interface to the Content Processor. Let us assume that maximum sized IEEE Ethernet frames are used for the bulk data transfer[30]. We also know that each Ethernet frame has an overhead of 20 bytes for inter packet gap[31] etc. We will therefore receive (10G / ((1500 + 20) x 8) packets per second, which is approximately 822,000 packet events per second.

Suppose that for each packet event received, we also receive a host read event to read the data contained in the TCP payload. Also suppose that this is a proxy connection, so the data is written back to another socket via a host write event. We assume that this data is acknowledged on the input stream. So, for each packet event received we also assume a host read event and a host write event. For bulk transfer on a 10Gbit full duplex interface we will therefore receive approximately 2,466,000 events per second.

2.8.3 Dispatcher Performance Based on the Timer Metric

The Astute Content Processor has a goal of servicing the timer expiration of 500,000 flows in 200ms. Servicing a timer event is similar to servicing a packet or host event, except that instead of issuing an LFKCREATE command to the FDC, we issue a SERVTIMER. We can therefore use the analysis of section 2.8.1 but instead of using 500K cps we use 2,500,000 timer events per second, i.e. if you replace step 1.a of section 2.8.1 with a SERVTIMER FDC command, then steps 1.a through 2.c must be executed 2,500,000 times per second.

3 Interfaces

3.1 Dispatcher LUC Bus

3.1.1 LUC Request Format

Figure 20 and Figure 21 illustrate the message formats used by the Dispatcher for sending requests to the LUC. The format to use depends upon the LUC command, as shown by Table 4.

[29] For more details on the events in a flow see "TCP Processing Paths for the Content Processor" and "Queuing Model Trace of a Simple HTTP Request" from section 1.1.
[30] Jumbo-sized Ethernet frames (16K) would put even less of a load on the Dispatcher.
[31] This is assuming an Ethernet. For some networks there may not be any inter packet gap, or it may be considerably less. However, this overhead is relatively small to the size of the frame, so the exact amount should not matter.

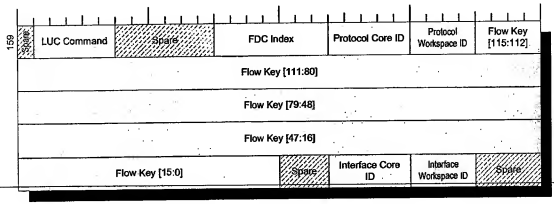

Figure 20: LUC Request Flow Key Format

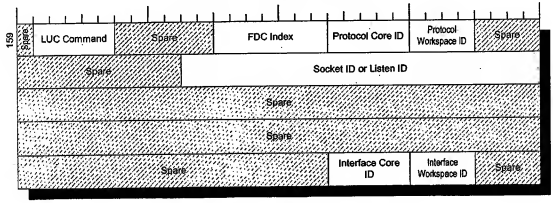

Figure 21: LUC Request Socket/Listen ID Format

Table 4 defines the commands that can appear in the LUC Command field of Figure 20 and Figure 21. For a complete description of these commands, including actual command values, the reader is referred to the *Lookup Controller (LUC) High Level Design* document.

Command	Description	LUC Format
RELS	Release any resources associated with this flow.	Flow Key Format
LFK	Lookup with Flow Key, do not create if not found.	Flow Key Format
LFKC	Lookup with Flow Key, create entry if not found.	Flow Key Format
LFLKC	Lookup with Flow Key. If not found then lookup with Listen Key: create an entry if Listen Key is found.	Flow Key Format
LLKC	Lookup with Listen Key, create entry if not found.	Flow Key Format
LFLK	Lookup with Flow Key. If not found then lookup with Listen Key. Do no create any entries.	Flow Key Format
LSID	Lookup with Socket ID.	Socket/Listen ID Format, using Socket ID
TDFK	Tear down with Flow Key.	Flow Key Format
TDLK	Tear down with Listen Key.	Flow Key Format
ULID	Update with Listen ID.	Flow Key Format
USID	Update with Socket ID.	Flow Key Format

Table 4: LUC Command Summary

Note that even though the ULID and USID LUC commands operate upon a Socket ID or Listen ID, the command format that we use is the Flow Key Format of Figure 20. For these commands, the LUC actually retrieves the Socket ID or Listen ID from the workspace that it receives from the Processor Cores.

3.1.2 Expected Performance

According to section 2.7, each packet or host event will cause two commands to be issued to the LUC from the Dispatcher. If we assume the connections per second metric, which should be the worst case, then each connection causes 8 such events, and we get 500K connections per second. We will therefore be issuing 8,000,000 LUC commands per second. From Figure 20 we can see that each LUC command requires 160-bits so the Dispatcher to LUC bus has a required bandwidth of 1.280Gbits per second. An 8-bit wide 266MHz bus would therefore be sufficient. However, to ease implementation we use a 32-bit wide Dispatcher to LUC Bus.

3.1.3 External Signals

The convention used for a signal name is <src>_<dst>_<name>.

Signal	# Of pins	I/O	Description
LUC_Disp_REQ	1	I	LUC is ready for command request from the Dispatcher.
Disp_LUC_DAT	32	O	LUC commands sent by Dispatcher.
Disp_LUC_VAL	1	O	Valid signal to envelop the data.

Table 5: Dispatcher LUC Bus Signals

3.1.4 LUC Request Queue

The *LUC Request Queue* of Figure 1 queues LUC commands that are issued by the Dispatcher on the Dispatcher LUC bus. This queue is sixteen entries deep.

3.1.4.1 Backpressure

When the *LUC Request Queue* passes a certain watermark, the Dispatcher stops servicing the IPU0, IPU1, SMC and LUC Timer interfaces. This backpressure is de-asserted once the queue drops below the watermark. The actual watermark value to use is defined in the LUC_ELMNT field of the BPRESSURE register (see section 3.14.13).

3.2 LUC Dispatcher Bus

3.2.1 LUC Timer Event Format

Figure 22 illustrates the format of the LUC Timer Event that is placed in the Timer Event buffer of Figure 1.

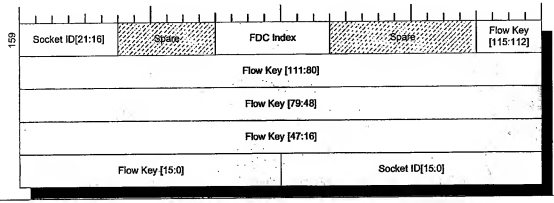

Figure 22: LUC Timer Event Format

3.2.2 Expected Performance

Timers are to be serviced at a rate of 500,000 per 200ms. This gives us 2,500,000 timer events per second. With reference to Figure 22, a timer event is 160-bits long. The LUC to Dispatcher bus therefore requires a bandwidth of 400Mbits per second. An 8-bit wide 266MHz bus would therefore be sufficient. However, to ease implementation we use a 32-bit wide LUC to Dispatcher Bus.

3.2.3 External Signals

The convention used for a signal name is <src>_<dst>_<name>.

Signal	# Of pins	I/O	Description
Disp_LUC_REQ	1	O	Dispatcher is ready for next the timer event from the LUC.
LUC_Disp_DAT	32	I	LUC SERVTIMER command sent by LUC.
LUC_Disp_VAL	1	I	Valid signal to envelop the data.

Table 6: LUC Dispatcher Bus Signals

3.3 Dispatcher FDC Bus

See the *Flow Director CAM (FDC) HLD* document for the format of the FDC commands and responses on this bus.

3.4 FDC Dispatcher Bus

See the *Flow Director CAM (FDC) HLD* document for the format of the FDC commands and responses on this bus.

3.5 IPU Dispatcher Bus

There are two of these buses in total, one for each IPU. Both buses follow the exact same protocol and bus formats. For signal details of the IPU to Dispatcher Bus the reader is directed to the *Input Processing Unit (IPU) HLD*.

Note that the servicing of both IPU buses can be disabled via the SMC_DISP_Almost_Full signal. See section 2.6.3 for further details.

3.5.1 Event Format

Figure 23 illustrates the format of events that appear on the IPU bus. It should be noted that this event has a hierarchical format, and that the only portion we define in this document is the *Control Event*, i.e. we define the portion of the event that the hardware needs to examine. For this reason, the total size of an event is not fixed: it depends on the type of event.

Note that not all fields of the Control Event of Figure 23 are valid. Those fields marked with horizontal faded lines are either filled in or adjusted by the Dispatcher, and are not valid when received from an IPU[32].

The format of Figure 23 is used on a number of buses in the Dispatcher. As described here, it is the format that is encountered on the IPU buses. It is also the same format that the Dispatcher will see on the SMC Bus and ND Event Queue. The Done Event Queue also uses this format, except only the first two 128-bit words are used.

For more details on the format of the data marked as *Event Dependent Data*, the reader is directed towards the *IPU to Processor Core Event Specification* document.

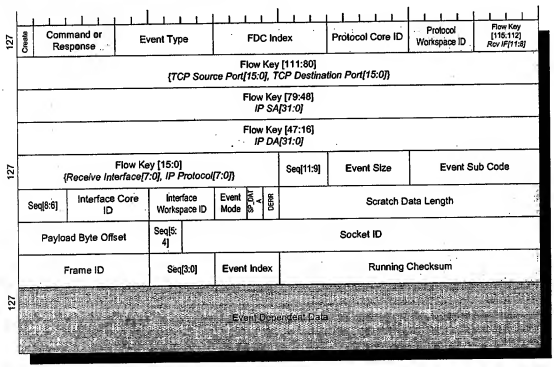

Figure 23: IPU Control Event Format

The following points should be noted about Figure 23:

1. Protocol done events are exactly two 128-bits long, i.e. it is a full Control Event but it has no Event Dependent Data. Note that done events **must** have a length of **exactly** two 128-bit words. If the done event is not this length then the Dispatcher will incorrectly process future done events.
2. The Event Dependent Data ends on a 128-bit word boundary. Even if the event frame format changes, this alignment will be kept.

[32] The fields marked with a cross-hatch pattern (Scratch Data Length and Running Checksum) are only modified by the Dispatcher for events received on the IPU0 and IPU1 interfaces.

Table 7 defines in more detail the various fields of Figure 23.

Field	Description
Create bit	Set to 1 to indicate that we should attempt a listen lookup, or create an entry, if the flow is not found in the LUC. The Packet Processor sets this field, e.g. when the SYN flag is set and only the SYN flag.
Command or Response	Used to indicate the various reasons as to why this event exists. Note that this field is only used for Dispatcher to Processor Core communication. The Processor Core need not, and should not, change this value. See Table 8.
Event Type	Indicates what type of event this is. The major types of events are Packet, Host, Timer and Done. Each of these events also has subtypes, e.g. TCP and UDP events are both packet events. Note that the actual values used are not important: the Dispatcher uses a level of indirection via registers.
FDC Index	Flow Director CAM Index. The Dispatcher sets field using the FDC response.
Protocol Core ID	The Protocol Core ID that has been assigned to process this event. The Dispatcher sets this field using the FDC response.
Protocol Workspace ID	Indicates where the flow state is being held in the Protocol Core. Note that not all events have an associated workspace. The type of command indicates whether this field is valid.
Flow Key	This is the 116-bit key that describes a flow.
Trace Sequence Number	This is a 12-bit sequence number that the Dispatcher places in each Type A message bus transaction. The 12-bits are split across the control event in four different locations. Note that this field is used for debugging only, and the operation of the Dispatcher in no way depends upon its value. See section 3.6.4 for further details.
Event Size	Number of 128-bit words in the event. Note that this is a 1's based value, i.e. value zero means there are zero 128-bit words in the event. For events that originate from the IPU, the Packet Processor sets the Event Size for Network Packets, and copies the Event Size for Host Messages. For events that originate from the Processor Cores, the SMC sets the Event Size.
Event Sub code	Used to distinguish between events that have the same event type. For example, the event type may say "Stateless processing", but the event sub code could say "ICMP" or "ARP" etc.
Interface Core ID	If two processor cores were assigned, then this field indicates the Interface Core ID that was selected. For stateless event types this field must be ignored.
Interface Workspace ID	If two processor cores were assigned then this field indicates where the flow state is being held in the Interface Core. Note that not all events have an associated workspace. The type of command indicates whether this field is valid. . For stateless event types this field must be ignored.
Event Mode	This is only used for the stateful done event that the processor core sends. The following values are valid: 00: Used to indicate that the Event Index is not valid, i.e. no event should be released. 01: Used to indicate that the Event Index field belongs to the protocol core. 10: Used to indicate that the Event Index field belongs to the interface core. 11: Not valid. Do not use. The Dispatcher sets this field to value 01 if the event is sent to a protocol core, and sets it to value 10 if the event is sent to an interface core. The processor cores can use this fact to avoid a modification of this field when they send the done event. Note that it is only used for stateful done events. For stateless done events it is always assumed that they are releasing the Event Index.
SP_DATA	This bit is set if the Frame ID is valid, i.e. it is set if data is held in the Scratchpad for this event.
DERR	The Dispatcher may modify this bit. If the Dispatcher encounters an error from the Scratchpad, or if it is requested to drop an event, then it sets this bit to one, otherwise it simply copies it from the input event. Note that the Packet Processor may have set this bit, so it is important to copy it and not set it to zero.
Scratch Data Length	Total number of bytes held in scratchpad memory that is associated with this event. The Dispatcher sets this field for events received from IPU0 or IPU1. The Dispatcher does not modify this field for events received from the ND event queue or the SMC.

Field	Description
Payload Byte Offset	Offset, in bytes, from the start of the first page of scratchpad memory to the first byte of data.
Socket ID	If the ACP is configured to support N flows, then the Socket ID will range from 0 to N-1. It is guaranteed to be unique for each flow.
Frame ID	This is the Frame ID in the Scratchpad where the frame data is being held.
Event Index	This is the value of the event index where the event was allocated. This is used to indicate which event can be released with the done event.
Running Checksum	This is a 1's compliment 16-bit checksum value. The Dispatcher adjusts this field to allow for data in the Scratchpad. Note that the Dispatcher only modifies this field for events received from IPU0 or IPU1. For events received from the ND event queue or from the SMC the Dispatcher does not modify this field. It should be noted that this checksum covers the entire packet. If padding was included in the packet then the software must correct this checksum. See section 2.5.1.2 for details.

Table 7: IPU Control Event Field Descriptions

Table 8 defines the values that can appear in the command or response field of Figure 23. Note that these values are only used for Dispatcher to Processor Core communication. The Processor Core need not, and should not, change these values when sending the Done event to the Dispatcher.

Field	Value (Hex)	Workspace	Description
COM_NONE	00	No	No command has been assigned. This should be the default value of the command / response field if no other value is specified.
COM_WITH_WS	01	Yes	Indicates that this is an event with an associated workspace ID.
COM_NO_WS	02	No	Indicates that this event does not have an associated workspace ID

Table 8: Command / Response Values

3.5.2 Expected Performance

Using the connections per second metric, we know that each connection consists of five frames from the network and three frames from the host[33]. Since our performance goal is 500,000 connections per second, we can expect 2,500,000 events per second on one IPU to Dispatcher Bus, and 1,500,00 events per second on the other IPU to Dispatcher Bus.

If we assume that the maximum size of a TCP based event is sixty-four 32-bit words then this gives a required bandwidth of approximately 5Gbits per second between an IPU and the Dispatcher for network traffic.

If we assume the same size for host based events, then this gives a required bandwidth of approximately 3Gbits per second between an IPU and the Dispatcher for host events.

3.5.3 Backpressure

The IPU Dispatcher Bus must allow backpressure to be exerted such that no input events are dropped.

3.6 Message Bus, Type A Transactions (To Protocol Cluster)

Message Bus Type A Transactions are used to send events to the Protocol Clusters. For more information on the Message Bus the reader is directed to the *Message Bus High Level Design* document.

[33] For more details on the events in a flow see "TCP Processing Paths for the Content Processor" and "Queuing Model Trace of a Simple HTTP Request" from section 1.1.

3.6.1 Event Format

The format of these events is exactly the same as Figure 23, except the fields marked with the horizontal lines would have been adjusted or filled in by the Dispatcher. Note that done events **must** have a length of **exactly** two 128-bit words. If the done event is not this length then the Dispatcher will incorrectly process future done events.

3.6.2 Addressing

The Dispatcher uses Type A message bus transactions to send to a specific Core ID using a specific Event Index. With reference to the Data Formats section of the *Message Bus High Level Design*, the Dispatcher forms an address for Type A transactions as follows:

1. The Core ID is placed in the Cluster[1:0] and Core[2:0] fields of the message bus address according to the format of Figure 6.
2. The WS/E bit of the message bus address is set to 0, which means this is an event.
3. The Queue Entry field of the message bus address is set to the Event Index.
4. The WS/V field of the message bus address is set to 2'b00, which means *Workspace Valid command from LUC disabled.*

3.6.3 Backpressure

Type A transactions are sent from the Dispatcher. Once the Message Bus has been arbitrated for, the source can **never** be held back.

3.6.4 Sequence Numbering

For the purpose of debugging, the Dispatcher inserts a 12-bit sequence number into every Type A transaction that it places on the Message Bus. This sequence number is initialised to zero at reset, and is incremented by one for each Type A message bus transaction that is sent. This sequence number wraps to zero when it reaches its maximum value. The software then uses this as a marker for any output events that it generates, allowing each input event to be easily matched with its corresponding output event.

Note that it is each Type A transaction that gets a unique sequence number, i.e. for a repeated unicast event each message that is sent out will get a unique sequence number.

The 12-bit sequence number is split across the event in four different sections. See Figure 23 for details.

3.7 *Message Bus, Type B Transactions (From Protocol Cluster)*

Message Bus Type B Transactions are used by the Protocol Cluster to send messages (*Done Events*) to the Dispatcher. For more information on the Message Bus the reader is directed to the *Message Bus High Level Design* document.

3.7.1 Event Format

The format of these events is exactly the same as Figure 23.

3.7.2 Addressing

The Dispatcher only receives Type B Message Bus transactions, so it does not create an address.

3.7.3 Done Event Queue

This queue is used by the Processor Cores to send Done Events. Done events have the same format as Figure 23 except unlike IPU events, all fields are valid. Note also that Done events only consist of the first two 128-bit words of Figure 23, i.e. Done events only consist of the Control Event format and do not contain any Event Dependent Data.

The Done Event Queue should allow for each Processor Core to issue a Done Event without causing backpressure. Since there are at most fifteen Processor Cores, this queue should be at least fifteen elements deep, i.e. 15 x 2 x 128-bits. To allow for the time for Processor Cores to react to the backpressure signal, we add another four elements. We then round up to the next even number, giving a Done Event Queue size of 20 x 2 x 128-bits.

3.7.3.1 Backpressure

Done events can **never** be dropped. The Dispatcher can send back an almost full signal to the Protocol Clusters that its queue is becoming full. Note however that the Protocol Clusters cannot back off immediately, so the Dispatcher must be prepared to receive some Done events even after it has asserted almost full.

The worst-case number of events that can be received after asserting almost full is created by the following sequence:

1. Each Protocol Cluster examines the almost full flag of the Dispatcher Done queue, and it is not set.
2. Each Protocol Cluster requests the Message Bus, which it cannot back out of.
3. The first Done event is sent and the Dispatcher raises almost full.
4. Assuming three Protocol Clusters, the Dispatcher must be prepared to receive at least another two Done events since two other Protocol Clusters have committed to the Message Bus.
5. Depending on how long it took the Dispatcher to raise (and propagate) the first almost full signal, the first Protocol Cluster may have scheduled another Done event.

Using this analysis we can see that about four Done events could be received after asserting almost full. To allow this value to be fine-tuned, the BPRESSURE register defines the exact value for when to assert backpressure via the DONE_ALMOST_FULL signal of Figure 1

3.8 SMC Dispatcher Bus

3.8.1 Event Format

The format of data on the SMC_Dispatcher_Bus is the same format as Figure 23, the IPU Control Event Format. As with the IPU, not all fields of this event have been filled in. In fact, the same fields that are invalid on IPU events are also invalid for the events on the SMC Dispatcher Bus.

3.8.2 Expected Performance

For the connections per second, bulk throughput and time metrics we do not expect this bus to be used.

3.8.3 External Signals

The convention used for a signal name is <src>_<dst>_<name>.

Signal	# Of pins	I/O	Description
SMC_Disp_Data	128	I	SMC to Dispatcher Data.
SMC_Disp_VAL_0	1	I	There is valid data from the first two quad-words on the data bus (part 1)
SMC_Disp_VAL_1	1	I	There is valid data from the remaining quad-words on the bus (part 2). Maximum of sixteen quad-words (256 bytes) will be transferred.
SMC_DISP_Almost_Full	1	I	Indicates that the SMC has passed a watermark in queuing Dispatcher events to DDR memory. See section 2.6.3 for how the Dispatcher interprets this signal.
Disp_SMC_RDY_0	1	O	Dispatcher is ready to receive first 2 quad-words from the event queue (part 1).
Disp_SMC_RDY_1	1	O	Dispatcher is ready to receive remaining quad-words of an event (part 2).
Disp_SMC_FRM_RPT	1	O	If high along with Disp_SMC_RDY_1, the data from part 2 will be repeated.

Table 9: SMC Dispatcher Bus Signals

Note that one cycle after the Dispatcher asserts SMC_Disp_RDY_1, the SMC_Disp_VAL_1 needs to be asserted along with valid data. This is because the Dispatcher is forwarding this event on the Message Bus, and the Message Bus cannot be stalled once a transfer begins. There is no such restriction on the Disp_SMC_RDY_0 signal.

The Disp_SMC_FRM_RPT signal is used when the Dispatcher is performing repeated unicast stateless processing. For this type of forwarding the Dispatcher needs to send the same event to multiple destinations.

3.8.4 Backpressure

The SMC Dispatcher Bus must allow backpressure to be exerted such that no input events are dropped.

3.9 MMC Bus

The reader is directed to the *ManageMent and Control HLD* for full details on the MMC bus.

3.10 Scratchpad Checksum Bus

There are two Dispatcher Scratchpad buses that are used for transferring checksum information, illustrated on Figure 1 as SP_BUS0 and SP_BUS1. These buses are used by the Dispatcher to query checksum, byte count and error information from the Scratchpad for a given Frame. The reader is directed to the Scratchpad HLD for full details on the Dispatcher Scratchpad Bus.

3.11 Scratchpad Reference Count Bus

The Scratchpad Reference Count Bus is illustrated on Figure 1 as SP_CNT_BUS. This bus is used for repeated unicast events where the Dispatcher must increase the reference count of a Scratchpad location. The reader is directed to the Scratchpad HLD for full details on the Scratchpad Reference Count Bus.

3.12 LUC FDC Bus

The LUC to FDC Bus provides an interface to the FDC for the LUC. The Dispatcher simply echoes the LUC_FDC_DAT to the FDC, and returns the FDC response on the FDC LUC Bus.

3.12.1 Expected Performance

For the connections per second metric, the LUC will issue an RMFIDX command for every event in the TCP connection. For the push model API there are eight events per TCP connection, and the required performance is 500K connections per second. The LUC FDC Bus will therefore see 4,000,000 RMFIDX commands per second. Each command is 128-bits wide, requiring a bandwidth of 512Mbits per second. The LUC to FDC bus is 32-bits wide at 266MHz, providing 8Gbits of bandwidth, which is ample.

3.12.2 External Signals

The convention used for a signal name is <src>_<dst>_<name>.

Signal	# Of pins	I/O	Description
LUC_FDC_DAT[31:0]	32	I	LUC to FDC Data.
LUC_FDC_VAL	1	I	LUC to FDC valid bit. This is a 4-cycle long envelop of LUC_FDC_DAT, since the FDC data (command) is 128-bits.
FDC_LUC_RDY	1	O	FDC is ready for the next LUC command.

Table 10: LUC FDC Bus Signals

3.13 FDC LUC Bus

The FDC to LUC Bus is used to send FDC responses back to the LUC. These responses are in result to FDC commands that are received on the LUC to FDC Bus (see section 3.12 above). With reference to the *Flow Director CAM HLD* the FDC response to the Dispatcher is 39-bits wide, but for the LUC FDC commands (CRTIMER and RMFIDX) not all bits are used. For this reason only the lower 26-bits of the FDC response are passed back to the LUC via the FDC LUC Bus.

3.13.1 Expected Performance

Using the same logic as section 3.12, there will be 4,000,000 responses to the RMFIDX command per second. Each response is 26-bits, which is a single transaction on the FDC LUC Bus. Since that bus is running at 266MHz there will be ample bandwidth.

3.13.2 External Signals

The convention used for a signal name is <src>_<dst>_<name>.

Signal	# Of pins	I/O	Description
FDC_LUC_DAT[25:0]	26	O	FDC to LUC Data. This represents the lower 26-bits of the 39-bit FDC response that the Dispatcher received.
FDC_LUC_VAL	1	O	Indicates that FDC_LUC_DAT is valid. This is a single cycle envelope.

Table 11: FDC LUC Bus Signals

3.14 Configuration Registers

In the following sections we define the microprocessor accessible registers that the Dispatcher contains.

3.14.1 Register Map

Table 12 defines the register map that the Dispatcher uses. The offsets in this table are relative to where the Dispatcher is assigned in the ManageMent and Control memory map. See the MMC HLD for further details.

3.14.2 Dispatcher Register Implementation

3.14.2.1 Write Access

During a Dispatcher register write, no ACK is supplied to the Management Controller (MMC). The Dispatcher must therefore be able to process back to back writes. According to the *Management Controller HLD*, the Dispatcher must be able to process back to back writes at a rate of one every 7 clock cycles.

3.14.2.2 Read Access

During a Dispatcher register read an ACK is supplied, so there is no issue regarding back to back reads.

Offset (Hex)	Mode	Register Name	Description
0000	Read Only	STATUS	Status information, e.g. error notification bits.
0001	Read / Write	CONTROL	Control information, e.g. reset bit.
0002	Read Only	FDC_ERR_CNT	Count of the number of error (5'b11111) from the FDC.
0003	Read Only	RCV_EVNT_CNT	Received event count.
0004	Read Only	FDC_NOSP_CNT	Count of the number of events processed where the FDC was full.
0005	Read Only	EPR_CNT	Count of the number of events that have been processed.
0006	Read Only	FDC_PEND_CNT	Count of the number of times we tried to update an entry that is in the FDC as *PENDING*.
0007	Read Only	FDC_TIMST_CNT	Count of the number of times we issued a LFKCREATE command but found the FDC entry in the *TIMER* state.
0008	Read / Write	MASK	Mask to apply before raising the MMC interrupt.
0009	Read / Write	EGPARMS	Event / Command Parameters that are not configured on a per event type basis.
000A	Read / Write	BPRESSURE	Parameters for when to apply backpressure.
000B	Read Only	FDC_DELST_CNT	Count of the number of times we tried to update an entry that is in the FDC as *DELETE*.
000C	Read / Write	DEBUG	Debug register.
000D-001F	N/A	SPARE	N/A
0020-003F	Read / Write	EPARMS	Various parameters that change the Dispatcher behaviour on a per Event Type basis.
0040-007F	N/A	SPARE	N/A
0080	FDC	FDC Register Block	This is the portion of the Dispatcher register block that is used to access the FDC registers. See section 3.14.17.

Table 12: Dispatcher Register Map

3.14.3 Status (STATUS) Register [0000H]

Indicates the status of the Dispatcher. Note that all error bits are reset when read.

Default value: 0

Bits	Name	Description
0	FDC_ERR	FDC Error bit. Set if the Dispatcher receives an error response from the FDC. Cleared on read.
1	DONE_OVERFLOW	This is set if the Done Queue overflows. This should never occur. If this bit gets set then events have been lost. Cleared on read.
2	SMC_DISP_AF_SET	This is a sticky version of the SMC_DISP_Almost_Full signal. It indicates if IPU0/1 servicing was ever disabled. Cleared on read.
3	DONE_BPRESS	This is set if the Done Queue is ever back pressured by the Dispatcher, i.e. set if the Done Queue ever passes its backpressure watermark. It is a sticky version of the DONE_ALMOST_FULL signal of Figure 1. Note that this bit does not get set when the DONE_AF_EN of the CONTROL register is set to 1. Cleared on read.
4	N/A	N/A.
5	LUC_BPRESS	This is set if the LUC Request Queue ever causes the Dispatcher to stop servicing the IPU0, IPU1, SMC and LUC interface, i.e. set if the LUC Request Queue ever passes its backpressure watermark. Cleared on read.
6	LUC_OVERFLOW	This is set if the LUC Request Queue ever overflows. This should never occur. If this bit gets set then LUC commands have been lost. Cleared on read.
7	INV_MMC_ADDR	This bit is set if the MMC attempts to access read an undefined register address on the Dispatcher or FDC. Note that this bit is not set when an undefined register is written to. Cleared on read.
8-31	N/A	N/A

Table 13: Status Register Bit Definitions

3.14.4 Control (CONTROL) Register [0001H]

Default value: 9'hff

Bits	Name	Description
0	IPU0_RDY_DIS	This is set to 1 at reset. It needs to be cleared before the IPU0 interface can be enabled. Setting this bit will disable the interface between the Dispatcher and IPU0. Setting this bit to 1 while the Dispatcher is running will cause it to process the current IPU0 event, but not request the next event.
1	IPU1_RDY_DIS	This is set to 1 at reset. It needs to be cleared before the IPU1 interface can be enabled. Setting this bit will disable the interface between the Dispatcher and IPU1. Setting this bit to 1 while the Dispatcher is running will cause it to process the current IPU1 event, but not request the next event.
2	SMC_RDY_DIS	This is set to 1 at reset. It needs to be cleared before the SMC interface can be enabled. Setting this bit will disable the interface between the Dispatcher and SMC. Setting this bit to 1 while the Dispatcher is running will cause it to process the current SMC event, but not request the next event.
3	SP_IPU0_REQ_DIS	This is set to 1 at reset. It needs to be cleared before the interface can be enabled. Setting this bit will disable the interface between the Dispatcher (IPU0 interface) and the Scratchpad. It is not recommended that this interface be disabled while the Dispatcher is running.
4	SP_IPU1_REQ_DIS	This is set to 1 at reset. It needs to be cleared before the interface can be enabled. Setting this bit will disable the interface between the Dispatcher (IPU1 interface) and the Scratchpad. It is not recommended that this interface be disabled while the Dispatcher is running.
5	LUC_RDY_DIS	This is set to 1 at reset. It needs to be cleared before the interface can be enabled. Setting this bit will disable the interface between the Dispatcher and the LUC. Setting this bit to 1 while the Dispatcher is running will cause it to process the current LUC event, but not request the next event.
6	FDC_LUC_RDY_DIS	This is set to 1 at reset. It needs to be cleared before the interface can be enabled. Setting this bit will disable the interface between the FDC and the LUC. It is not recommended that this interface be disabled while the Dispatcher is running.

Bits	Name	Description
7	DONE_AF_EN	This is set to 1 at reset. It needs to be cleared before the interface can be enabled. Setting this bit will disable the interface between the Dispatcher and the Done Queue on the Message Bus. Setting this bit to 1 while the Dispatcher is running will cause it to assert backpressure on the Message Bus.
9	DIS_SMC_DISP_BP	This is the *Disable SMC-DISP Backpressure* bit. If this bit is set to zero then the SMC_DISP_Almost_Full signal is used to determine whether to service IPU0/1 events. If this bit is set to one then IPU0/1 events are serviced regardless of the SMC_DISP_Almost_Full signal. See section 2.6.3 for further details.
8-31	N/A	N/A

Table 14: Control Register Bit Definitions

3.14.5 FDC Error Count (FDC_ERR_CNT) Register [0002H]

This register counts the number of times we got an error (5'b11111) response from the FDC. Since events that get error responses are left on their input queue, this counter may count the same event multiple times. However, since this counter is only used for debug this behaviour is acceptable.

Default value: 0

Bits	Name	Description
0-31	CNT	Number of times we got an error response (5'b11111) from the FDC. Cleared on read.

Table 15: FDC No Space Count Register Bit Definitions

3.14.6 Received Event Count (RCV_EVNT_CNT) Register [0003H]

This register counts the number of times the Dispatcher has attempted to service an event. Since events that cannot be serviced are left on their input queue, this counter may count the same event multiple times. This will be most apparent when there are no other events to service other than the one that is blocked. However, since this counter is only used for debug this behaviour is acceptable.

Default value: 0

Bits	Name	Description
0-31	CNT	Number of events that the Dispatcher has attempted to service. Cleared on read.

Table 16: Received Event Count Register Bit Definitions

3.14.7 FDC No Space Count (FDC_NOSP_CNT) Register [0004H]

This register counts the number of times the Dispatcher has attempted to service an event but failed due to a lack of space in the FDC. Since events that cannot be serviced are left on their input queue, this counter may count the same event multiple times. This will be most apparent when there are no other events to service other than the one that is blocked. However, since this counter is only used for debug this behaviour is acceptable.

Default value: 0

Bits	Name	Description
0-31	CNT	Number of events that received a "no space" response from the FDC. Cleared on read.

Table 17: FDC No Space Count Register Bit Definitions

3.14.8 Events Processed Count (EPR_CNT) Register [0005H]

This register counts the number of events that have been completely serviced by the Dispatcher, i.e. removed from their input queue.

Default value: 0

Bits	Name	Description
0-31	CNT	Number of events that have been completely processed by the Dispatcher. Cleared on read.

Table 18: Events Processed Count Register Bit Definitions

3.14.9 FDC In Pending Count (FDC_PEND_CNT) Register [0006H]

This register counts the number of times the Dispatcher has attempted to service an event but failed due to the FDC entry being in the *PENDING* state. Since events that cannot be serviced are left on their input queue, this counter may count the same event multiple times. This will be most apparent when there are no other events to service other than the one that is blocked. However, since this counter is only used for debug this behaviour is acceptable.

Default value: 0

Bits	Name	Description
0-31	CNT	Number of time we tried to update a flow and found its FDC entry in the *PENDING* state. Cleared on read.

Table 19: FDC In Pending Count Register Bit Definitions

3.14.10 FDC In Timer State (FDC_TIMST_CNT) Register [0007H]

This register counts the number of times the Dispatcher issued a LFKCREATE command but found that the FDC entry was in the *TIMER* state. Since events that cannot be serviced are left on their input queue, this counter may count the same event multiple times. However, since this counter is only used for debug this behaviour is acceptable.

Default value: 0

Bits	Name	Description
0-31	CNT	Count of the number of times we issued a LFKCREATE command but found the FDC entry in the *TIMER* state.

Table 20: FDC In Timer State Register Bit Definitions

3.14.11 Mask (MASK) Register [0008H]

Default value: 0

Bits	Name	Description
0-31	MASK	Mask to apply to the STATUS register before determining whether to raise the interrupt line to the MMC. Not all bits in this register are used. Only the bits that have corresponding defined bits in the STATUS register are used.

Table 21: Mask Register Bit Definitions

3.14.12 Event / Command Parameters (ECPARMS) Register [0009H]

The ECPARMS register contains event and command parameters that are not configured on a per event type basis.

Default value: 0

Bits	Name	Description
0-4	ETSERV	The Event Type value to use for a SERVTIMER command to the LUC. Note that this must be a stateful event, i.e. the MSB must be zero. For this reason we only store 5-bits of Event Type.
5-31	N/A	N/A

Table 22: Event / Command Parameters Register Bit Definitions

3.14.13 Backpressure (BPRESSURE) Register [000AH]

The BPRESSURE register allows us to program the watermarks for when to assert backpressure on various interfaces. This register contains values for the Done Event Queue and the LUC Request Queue. Note that they refer to the number of elements in the queue, and not the number of bytes.

Default value: See field descriptions

Bits	Name	Description
0-4	DNUM_ELMNT	Number of *elements* to allow into the Done Queue before the DONE_ALMOST_FULL signal of Figure 1 is asserted. Backpressure is asserted when greater than or equal to DNUM_ELMNT *elements* are in the queue. Backpressure is not asserted if less than DNUM_ELMNT *elements* are queued. Default value: 5'h10
5-8	N/A	N/A.
9-12	LUC_ELMNT	Number of *elements* (LUC commands) to allow in the *LUC Request Queue* of Figure 1 before the Dispatcher stops servicing the IPU0, IPU1, SMC and LUC Timer interfaces. Backpressure is asserted when greater than or equal to LUC_ELMNT *elements* are in the queue. Backpressure is not asserted if less than LUC_ELMNT *elements* are queued. Default value: 4'hb
13-31	N/A	N/A

Table 23: Message Bus Backpressure Register Bit Definitions

3.14.14 FDC In Delete State (FDC_DELST_CNT) Register [000BH]

This register counters the number of times the Dispatcher issued a LFKCREATE command but found that the FDC entry was in the *DELETE* state. Since events that cannot be serviced are left on their input queue, this counter may count the same event multiple times. However, since this counter is only used for debug this behaviour is acceptable.

Default value: 0

Bits	Name	Description
0-31	CNT	Count of the number of times we issued a LFKCREATE command but found the FDC entry in the *DELETE* state.

Table 24: FDC In Delete State Register Bit Definitions

3.14.15 Debug (DEBUG) Register [000CH]

This debug register is shared between the Dispatcher and the FDC.

Default value: 0

Bits	Name	Description
0-5	MUX_SEL	Debug multiplexer select. This selects what debug information is exported via the debug bus. When set to zero the Dispatcher/FDC debug input is connected to the debug output, i.e. the Dispatcher/FDC is bypassed. When set to a non-zero value the Dispatcher/FDC will present debug information to the debug output. See the Dispatcher/FDC implementation documentation for details.
6-31	N/A	N/A

Table 25: Debug Register Bit Definitions

3.14.16 Event Parameter (EPARMS) Registers [0020H – 003FH]

The Event Parameter registers block is a set of registers that contain forwarding information for a specific event type. This register block is indexed by the lower 5-bits of the Event Type. There are thirty-two Event Parameter registers. The reason there are only thirty-two Event Parameter registers even though the Event Type is 6-bits is because if the most significant bit of an Event Type is set then the event requires Unicast Event Processing. This is in accordance with the Event Type substructure defined in section 1.4.4 and in the *Flow Director CAM HLD*. If the most significant bit of the Event Type is not set then we use the lower 5-bits to index into the EPARMS register block. The FW_MODE then tells us whether this is a repeated unicast, unicast with drop, stateful or unicast event.

Note that the format of the lower 15 bits depends upon the value of the FW_MODE field. If the FW_MODE has value 2'b10 then the LUCCM1, LUCCM2, DONE_TD and LUSEFK fields are valid and can be used. If the FW_MODE has value 2'b00 then the CREPEAT field is valid and can be used. For other values of FW_MODE bits, EPARMS[0:14] and EPARMS[17:31] are not valid.

Default value: 0

	Bits	Name	Description
FW_MODE has value 2'b10	0-4	LUCCM1	LUC Command 1. Used when the create flag is turned on, or for tearing down entries.
	5-9	LUCCM2	LUC Command 2. Used when the create flag is not on, or for updating entries.
	10	DONE_TD	Event Type is a "Done, please Teardown".
	11	LUSEFK	Set to 1 if the LUCCM2 Command should use a Flow Key. Set to 0 if the LUCCM2 Command should use a Socket ID.
	12-14	N/A	N/A
FW_MODE has value 2'b00	0-14	CREPEAT	Bitmap of Processor Cores that must receive messages of this type. This is in Core Bitmap format, as described by section 2.3.3. Note that this field is only used for Repeated Unicast Event Processing. Also note that it is an invalid configuration to have the FW_MODE set to Repeated Unicast, and the CREPEAT field set to zero.
	15-16	FW_MODE	This is the forwarding mode for events of this type. With reference to section above, the values of this field are defined as: 2'b00: Repeated Unicast Stateless Event Processing 2'b01: Unicast Stateless Event Processing, Drop Condition 2'b10: Stateful Event Processing 2'b11: Unicast Stateless Event Processing
	17-31	N/A	N/A

Table 26: Event Parameter Register Bit Definitions

3.14.17 FDC Register Block [0080H – 00FFH]

As described previously, the FDC and the Dispatcher share the same ManageMent and Control (MMC) interface. The FDC register block is therefore a block of 128 registers that the FDC unit will respond to. For the exact mapping of these 128 registers the reader is referred to the register map in the FDC HLD.

3.15 Initialisation

The following sequence should be used to initialise the Dispatcher from the reset state:

1. Fully initialise the FDC. See the initialisation section of the *Flow Director CAM HLD* for further details.

2. Program the following Dispatcher registers with values appropriate to the software:
   a. Mask (MASK) register (section 3.14.11).
   b. Event / Command Parameters (ECPARMS) register (section 3.14.12).
   c. Backpressure (BPRESSURE) register (section 3.14.13).
   d. Event Parameter (EPARMS) register block (section 3.14.16). For event types that are not being used, it is suggested that the forwarding mode is set to *Unicast Stateless Event Processing, Drop Condition*. Note: this still requires a valid (non-empty) event mask in the FDC.

3. Clear the IPU0_RDY_DIS and IPU1_RDY_DIS bits of the CONTROL register. These are set to 1 by default and need to be set to 0 to enable the interface between the Dispatcher and IPU0 / IPU1.

4. Clear the SMC_RDY_DIS bit of the CONTROL register. This is set to 1 by default. When set to 0 this enables the interface between the SMC and the Dispatcher.

5. Clear the SP_IPU0_REQ_DIS and SP_IPU1_REQ_DIS bits of the CONTROL register. These are set to 1 by default and need to be set to 0 to enable the two interfaces between the Dispatcher and the Scratchpad for IPU0 and IPU1.

6. Clear the LUC_RDY_DIS bit of the CONTROL register. This is set to 1 by default. When set to 0 this enables the interface between the Dispatcher and the LUC.

7. Clear the FDC_LUC_RDY_DIS bit of the CONTROL register. This is set to 1 by default. When set to 0 this enables the interface between the LUC and the FDC.

8. Clear the DONE_AF_EN bit of the CONTROL register. This is set to 1 by default. While set to 1 this forces the DONE_ALMOST_FULL signal of Figure 1 to high, effectively disabling the interface between the Message Bus and the Dispatcher for Done Events. Clearing this signal enables that interface, and allows DONE_ALMOST_FULL to assert backpressure when required.

9. Done. The Dispatcher is now initialised.

4 Open Issues

None.

5 Summary

The preceding sections should have accurately described the operation of the Dispatcher. Please notify the author of any discrepancies, omissions or typos.

6 External References

[RFC 1071] Computing the Internet Checksum, R. T. Braden and J. B. Postel. See http://intrastute/users/brian/rfc/rfc1071.txt for a local copy.